R O B O T R O N

# SYSTEMHANDBUCH SCP

Anleitung fuer den Programmierer

Teil II - Assemblerprogrammierung

Stand: April 1985

VEB Robotron
Bueromaschinenwerk Soemmerda

Die vorliegende 1. Auflage der - Dokumentation "Anleitung fuer den Programmierer" - entspricht dem Stand April 1985 und unterliegt nicht dem Aenderungsdienst.

Nachdruck, jegliche Vervielfaeltigung oder Auszuege daraus sind unzulaessig.

Die Dokumentation wurde durch ein Kollektiv des VEB Robotron Bueromaschinenwerk Soemmerda ausgearbeitet.

Im Interesse einer staendigen Weiterentwicklung werden die Leser gebeten, dem Herausgeber ihre Vorschlaege bzw. Hinweise zur Verbesserung mitzuteilen.

VEB Robotron
Bueromaschinenwerk Soemmerda
Bereich Forschung und Entwicklung

5230 Soemmerda

Weissenseerstr. 52

Seite

1. Assembler ASM 7

1.1. Einleitung 7

1.2. Ueberblick 7

1.3. Quelldatei-Organisation 8
1.3.1. Formate 8
1.3.1.1. Quellzeilen-Format 8
1.3.1.2. Kommentar 9
1.3.2. Symbole 9
1.3.2.1. Label (Marke) 9
1.3.2.2. Public 10
1.3.2.3. External 11
1.3.2.4. Speicherzuordnungszaehler-Modus 12
1.3.3. Operationskodes und Pseudooperationen 13
1.3.4. Argumente (Ausdruecke) 14
1.3.4.1. Operanden 15
1.3.4.2. Operatoren 19

1.4. Assembler-Eigenschaften 22
1.4.1. Einzelfunktions-Pseudooperationen 22
1.4.1.1. Pseudooperationen zur Auswahl der Anweisungsliste 22
1.4.1.2. Pseudooperationen zur Datei- und Symboldefinition 23
1.4.1.3. Pseudooperation zur Zuweisung des Speicherplatzzuordnungszaehler-Modus 28
1.4.1.4. Dateibezogene Pseudooperationen 32
1.4.1.5. Pseudooperationen zur Listensteuerung 35
1.4.2. Makrofaehigkeit 41
1.4.2.1. Pseudooperationen zur Makrodefinition 42
1.4.2.2. Wiederholungs-Pseudooperationen 44
1.4.2.3. Beendigungs-Pseudooperationen 47
1.4.2.4. Pseudooperation fuer Makro-Symbole 48
1.4.2.5. Spezielle Makro-Operatoren 48
1.4.3. Pseudooperationen zur bedingten Assemblierung 50

1.5. Abarbeitung des Assemblers 52
1.5.1. Aufrufen des Assmenblers 52
1.5.2. Die Kommandozeile des Assemblers 52
1.5.2.1. Source (=dateiname) 54
1.5.2.2. Objekt (dateiname) 55
1.5.2.3. List (,dateiname) 55
1.5.2.4. Schalter (/switch) 56
1.5.2.5. Zusaetzliche Angaben der Kommandozeile 58
1.5.3. Format des Assemblerprotokolls 62
1.5.3.1. Format der Assemblerliste 62
1.5.3.2. Format der Symboltabelle 63

Seite

1.5.4. Fehlerkodes und Fehlermeldungen 64
1.5.4.1. Fehlerkodes 64
1.5.4.2. Meldungen 66

1.6. Beschreibung der Befehle 67
1.6.1. Ladebefehle 67
1.6.1.1. 8-Bit-Ladebefehle 67
1.6.1.2. 16-Bit-Ladebefehle 69
1.6.2. Indirekte Registeroperationen 71
1.6.2.1. PUSH-Befehle 71
1.6.2.2. POP-Befehle 72
1.6.3. Register-Austausch-Befehle 73
1.6.4. Blocktransportbefehle 75
1.6.5. Blocksuchbefehle 76
1.6.6. Arithmetische und logische Operationen 77
1.6.6.1. 8-Bit-Arithmetik 77
1.6.6.2. 16-Bit-Arithmetik 80
1.6.6.3. 8-Bit-Logikbefehle 81
1.6.7. Sprungbefehle 82
1.6.8. Verschiebebefehle 83
1.6.9. Spezielle Akkumulator- und Flagbefehle 86
1.6.10. Unterprogramm-Aufruf-Befehle 87
1.6.11. Unterprogramm-Ruecksprung-Befehle 89
1.6.12. CPU-Steuerbefehle 90
1.6.13. Bittest- und -Setzbefehle 91
1.6.14. Eingabebefehle 92
1.6.15. Ausgabebefehle 94
1.6.16. Abkuerzungsverzeichnis zur Befehlsbeschreibung 95
1.6.17. Arbeit mit den Bedingungsbits 96

1.7. Cross-Referenz 98
1.7.1. Erzeugen einer Cross-Referenz-Liste 99
1.7.1.1. Erzeugen einer Cross-Referenz-Datei 99
1.7.1.2. Generieren einer Cross-Referenz-Liste 99
1.7.2. Pseudooperationen zur Listensteuerung 99

1.8. Uebersicht ueber die Pseudooperationen des Assemblers 101

1.9. Mikrobefehlsliste - Z80-Operationskodes 104

1.10. Mikrobefehlsliste - 8080-Operationskodes 135

2. LINK 147

2.1. Aufruf des Programmverbinders LINK 149

2.2. LINK-Kommandos 149
2.2.1. Dateinamen 150
2.2.2. Schalter 151

2.3. Fehlermeldungen 162

Seite

| | | |
|---|---|---|
| 3. | LIB | 165 |
| 3.1. | Aufruf des Bibliothekars | 165 |
| 3.2. | Anwendungsfaelle | 166 |
| 3.2.1. | Bilden einer Bibliothek | 166 |
| 3.2.2. | Listen einer Bibliothek | 166 |
| 3.3. | LIB-Kommandos | 167 |
| 3.3.1. | Zielfeld | 167 |
| 3.3.2. | Quellfeld | 167 |
| 3.3.3. | Schalterfeld | 170 |
| 4. | Beschreibung des Debuggers | 172 |
| 4.1. | Aufgaben | 172 |
| 4.2. | Aufruf des Programmes und Belegung des TPA | 172 |
| 4.3. | Aufbau der Debugger-Kommandozeilen | 178 |
| 4.4. | Angabe der Parameter | 179 |
| 4.4.1. | Operanden | 179 |
| 4.4.1.1. | Hexadezimale Zahlen | 179 |
| 4.4.1.2. | Dezimale Zahlen | 179 |
| 4.4.1.3. | Zeichenkettenwerte | 180 |
| 4.4.1.4. | Symbolische Bezuege | 180 |
| 4.4.1.5. | Bedingte Symbolreferenzen | 181 |
| 4.4.1.6. | CPU-Register | 182 |
| 4.4.1.7. | Der Ausdrucksakkumulator als Operand | 184 |
| 4.4.1.8. | Stapeloperanden | 183 |
| 4.4.2. | Parameterwerte | 184 |
| 4.5. | Uebersicht ueber Syntax der Debugger-Kommandos | 185 |
| 4.6. | Beschreibung der Kommandos | 186 |
| 4.6.1. | Uebersicht ueber Funktionen | 186 |
| 4.6.2. | Dateiarbeit | 187 |
| 4.6.2.1. | Uebergabe von CCP-Parametern (I-Kommando) | 187 |
| 4.6.2.2. | Einlesen von Dateien (R-Kommando) | 188 |
| 4.6.3. | Dateimanipulation im Arbeitsspeicher | 191 |
| 4.6.3.1. | Anzeigen eines Speicherbereiches (D-Kommando) | 191 |
| 4.6.3.2. | Fuellung des Speichers (F-Kommando) | 192 |
| 4.6.3.3. | Modifikation von Speicherstellen (S-Kommando) | 193 |
| 4.6.3.4. | Blocktransfer im Arbeitsspeicher (M-Kommando) | 194 |
| 4.6.3.5. | Vergleich von zwei Datenbloecken (E-Kommando) | 194 |
| 4.6.3.6. | Suche einer Kette (K-Kommando) | 195 |
| 4.6.4. | Assembler und Reassembler | 196 |
| 4.6.4.1. | Assembler (A-Kommando) | 196 |
| 4.6.4.2. | Reassembler (L-Kommano) | 197 |
| | | Seite |
| 4.6.5. | Hilfsfunktionen (H-Kommando) | 197 |
| 4.6.6. | Aufruf von Systemunterprogrammen | 198 |

(C-Kommando)
4.6.7. Echtzeittest von Programmen 199
4.6.7.1. Anzeige und Modifikation der CPU-Register 199
(X-Kommando)
4.6.7.2. Modifikation von Passpunkten (P-Kommando) 200
4.6.7.3. Uebergang in den Echtzeitbetrieb 201
(G-Kommando)
4.6.7.4. Schrittweise Abarbeitung mit Spur 202
(T-Kommando)
4.6.7.5. Schrittweise Abarbeitung ohne Spur 203
(U-Kommando)

4.7. Erweiterungen des Debuggers 204
4.7.1. Arbeit mit den Erweiterungen 204
4.7.2. Das Dienstprogramm HIST 205
4.7.3. Das Dienstprogramm TRACE 206

# 1. Assembler ASM

## 1.1. Einleitung

Der Assembler benoetigt 19 K Speicherplatz plus 4 K Pufferbereich.

Syntax-Notation:

| | |
|---|---|
| [ ] | Die Eingabe ist optional (wahlfrei). |
| < > | Ein Text in Kleinbuchstaben in spitzen Klammern muss durch den Anwender bereitgestellt werden (z.B. <dateiname>). Ein Text in Grossbuchstaben gibt eine Tastenbedienung an (z.B. <ET>). |
| { } | Auswahl zwischen 2 oder mehreren Eintragungen |
| ... | Die Eintragung kann wiederholt werden. |

Alle anderen Interpunktionszeichen , : = / * $ . werden unveraendert uebernommen.

## 1.2. Ueberblick

a) Der Assembler unterstuetzt 2 Assemblersprachen:

- 8080-Mnemonik
- Z80-Mnemonik

b) Der Assembler erzeugt relativen und/oder absoluten Kode.

c) Der Assembler unterstuetzt das Schreiben von Makros.
Der Programmierer schreibt einen Block fuer eine Anzahl Anweisungen. Dieser Block erhaelt einen Namen, mit dem der Makro aufgerufen wird. Die Anweisungen sind die Makro-Definition.
Wenn diese Befehle an irgendeiner Stelle benoetigt werden, ruft der Programmierer den Makro auf. Der Makro-Aufruf uebergibt ausserdem die Parameter an den Assembler fuer die Makro-Expansion.
Das Verwenden von Makros reduziert den Umfang fuer eine Quelle, da die Makro-Definitionen in einer gesonderten Datei auf der Diskette stehen koennen und nur in den Modul uebernommen werden, wenn sie waehrend der Assemblierung gefordert werden.
Makros koennen geschachtelt werden, d.h. in einem Makro kann ein anderer aufgerufen werden. Die Schachtelungstiefe ist nur durch den Speicherplatz begrenzt.

d) bedingte Assemblierung
Der Assembler unterstuetzt die bedingte Assemblierung. Der Programmierer kann eine Bedingung bestimmen, unter welcher Teile des Programms entweder assembliert oder nicht assembliert werden.
Die Moeglichkeiten der bedingten Assemblierung werden durch einen kompletten Satz bedingter Pseudooperationen vergroessert, welche das Testen der Assembler-Paesse, der Symbol-Definitionen und der Makro-Parameter einschliessen.
Die Bedingungen koennen bis zu 255-mal geschachtelt werden.

## 1.3. Quelldatei-Organisation

### 1.3.1. Formate

Eine Assembler-Quelldatei ist eine Anzahl von Zeilen, die in Assembler-Sprache geschrieben sind.
Die letzte Zeile der Datei muss eine END-Anweisung sein, die durch <ET> abgeschlossen ist.
Entsprechende Anweisungen, wie z.B. IF...ENDIF, muessen in der richtigen Reihenfolge stehen. Anderenfalls koennen die Zeilen in beliebiger Reihenfolge des Programmentwurfs erscheinen.

#### 1.3.1.1. Quellzeilen-Format

Eine Eingabe-Quelldatei fuer den Assembler besteht aus Anweisungs-Zeilen, die in Teile oder Felder unterteilt sind:

| **Symbol** | **Operation** | **Argument** | **Kommentar** |
|---|---|---|---|

Symbol — Dieses Feld kann eine der drei Symboltypen
. Label (Marke)
. Public
. External
enthalten; gefolgt von einem Doppelpunkt, ausser wenn das Symbol Teil einer SET- ,EQU- oder MAKRO-Anweisung ist.

Operation — Dieses Feld enthaelt einen Operationskode, eine Pseudooperation, einen Makro-Namen oder einen Ausdruck.

Argument — Dieses Feld enthaelt Ausdruecke, Variable, Register-Namen, Operanden und Operatoren.

Kommentar — Dieses Feld enthaelt Kommentartext, der immer mit einem Semikolon beginnen muss.

Alle Felder sind optional. Auch Leerzeilen sind moeglich.
Die Anweisungen beginnen in einer beliebigen Spalte. Mehrere Leerzeichen oder Tabulatoren zwischen den Feldern koennen zur besseren Lesbarkeit eingefuegt werden, aber mindestens ein Leerzeichen oder Tabulator ist zwischen zwei Feldern noetig.

### 1.3.1.2. Kommentar

Ein Kommentar beginnt immer mit einem Semikolon und endet mit <ET>.
Fuer lange Kommentare kann die .COMMENT-Pseudooperation (siehe Pkt. 1.4.1.4.) verwenden werden, um das Einfuegen des Semikolons auf jeder Zeile des Kommentars zu vermeiden.

### 1.3.2. Symbole

Symbole sind Namen fuer Teilfunktionen oder Werte. Symbolnamen werden durch den Programmierer definiert. Die Symbole fuer dieses Programmpaket gehoeren zu einem von drei Typen, gemaess ihrer Funktion.
Diese 3 Typen sind:

- Label (Marken)
- Public
- External

Alle 3 Typen haben ein Modus-Attribut, das mit dem Speichersegment korrespondiert, welches das Symbol repraesentiert (siehe Pkt. 1.3.2.4.).

Alle 3 Typen der Symbole haben folgende Charakteristik:

1. Symbole koennen beliebige Laenge haben, aber die Anzahl der signifikanten Zeichen, die an den Programmverbinder uebergeben werden, variiert mit dem Typ des Symbols:

   a) Fuer Labels sind die ersten 16 Zeichen signifikant.
   b) Fuer Public- und External-Symbole werden die ersten 6 Zeichen an den Programmverbinder uebergeben.

   Zusaetzliche Zeichen werden "intern" abgeschnitten.

2. Ein gueltiges Symbol kann folgende Zeichen enthalten:
   A - Z  0 - 9  $ . ? @ _

3. Ein Symbol darf nicht mit einer Ziffer oder einem Unterstreichungszeichen beginnen.

4. Beim Lesen eines Symbols werden Kleinbuchstaben in Grossbuchstaben gewandelt. Es koennen also auch Klein- und Grossbuchstaben in einem Symbol gemischt auftreten.

### 1.3.2.1. Label (Marke)

Ein Label ist ein Referenzpunkt fuer eine Anweisung **innerhalb** des Programmoduls, in dem diese Marke definiert ist.
Eine Marke setzt ihren Adresswert der Symbolmarke auf die Adresse der Daten, die der Marke zugeordnet sind.

Beispiel: BUFF: DS 1ØØØH

BUFF ist gleich der 1. Adresse der 1ØØØH Bytes des reservierten Platzes.
Ist eine Marke einmal definiert, kann sie als Eintragung im Argumentfeld verwendet werden.
Eine Anweisung mit einer Marke in ihrem Argument bezieht sich auf die Anweisungszeile, in welcher diese Marke im Symbolfeld als Marke definiert ist.

Beispiel: LD A,(BUFF)

schreibt den Inhalt des A-Registers auf den Speicherplatz, den die Marke BUFF definiert.

Eine Marke kann ein beliebiges gueltiges Symbol bis zu 16 Zeichen Laenge sein.
Wenn eine Marke definiert werden soll, so muss sie als erstes in der Anweisungszeile stehen.
Sowohl bei Z80- als auch bei 8080-Mnemonik muss nach einer Marke sofort **ein** Doppelpunkt (kein Leerzeichen oder Tabulator) folgen; es sei denn, die Marke ist Teil einer SET- oder EQU-Anweisung (dann kein Doppelpunkt).
(Folgen der Marke 2 Doppelpunkte, dann wird sie zum Public-Symbol!)

### 1.3.2.2. Public

Ein Public-Symbol ist als Marke definiert. Der Unterschied zur Marke besteht darin, dass ein Public-Symbol auch als Referenzpunkt **in anderen Programmoduln** verfuegbar ist.

Es gibt 2 Moeglichkeiten, ein Symbol als Public zu erklaeren:

1. 2 Doppelpunkte (::) folgen dem Namen
   z.B. MARKE:: RET

2. Verwendung einer der Pseudo-Operationen PUBLIC, ENTRY oder GLOBAL

```
PUBLIC MARKE
```

Beispiel:
Das Resultat beider Moeglichkeiten ist identisch:

```
MARKE:: RET
```

ist aequivalent zur Anweisungsfolge

```
PUBLIC  MARKE
.
.
MARKE:  RET
```

### 1.3.2.3. External

Ein externes Symbol ist ausserhalb des aktuellen Programmoduls in einem anderen, separaten Modul als Public-Symbol definiert.

Die Wertzuweisung erfolgt waehrend der Programmverbindung. Dabei wird dem externen Symbol der Wert des Public-Symbols aus dem anderen Programmodul uebergeben.

Beispiel:

```
MODUL1

MARKE::    DB          7          ;PUBLIC  MARKE=7

MODUL2

           BYTE EXT    MARKE      ;EXTERNAL MARKE
```

Waehrend der Programmverbindung wird fuer EXTERNAL MARKE der Wert von PUBLIC MARKE verwendet.

Ein Symbol wird zum External erklaert durch:

1. 2 Doppelkreuze (##) nach dem Namen des Symbols.

   Beispiel: `CALL MARKE##`

   erklaert MARKE als ein Symbol mit einem 2-Byte-Adresswert, der in einem anderen Modul definiert ist.

2. Fuer 2-Byte-Werte eine der Pseudo-Operationen

   EXT, EXTRN oder EXTERNAL

   Beispiel: `EXT MARKE`

   erklaert MARKE als ein 2-Byte-Symbol, der in einem anderen Modul definiert ist.

3. Fuer 1-Byte-Werte eine der Pseudooperationen

   BYTE EXT ,BYTE EXTRN oder BYTE EXTERNAL

   Beispiel: `BYTE EXT MARKE`

   erklaert MARKE als 1-Byte-Wert, der in einem anderen Programmodul definiert ist.

Das Ergebnis aller 3 Moeglichkeiten der Vereinbarung ist dasselbe.

```
CALL  MARKE##     ist aequivalent zur Anweisungsfolge

EXT   MARKE
CALL  MARKE
```

### 1.3.2.4. Speicherzuordnungszaehler-Modus (Typenarten)

Ein Symbol bezieht sich beim Auftreten seines Namens im Argumentfeld auf eine Anweisungszeile.
Der Wert des Symbols ist die Adresse der Anweisung, die durch das Symbol markiert wird. Wenn das Symbol im Argumentfeld auftritt, wird es durch seinen Wert ersetzt und in der Operation verwendet.
Der Wert eines Symbols wird gemaess seinem Speicherzuordnungszaehler (PC)-Modus bestimmt.

Durch den Speicherzuordnungszaehler-Modus ist eine Teilung des Speichers in 4 Segmente moeglich. Er bestimmt, in welches Segment der Programmteil geladen wird.
Die 4 Segmente sind:

- . absolutes Segment
- . datenrelatives Segment
- . koderelatives Segment
- . commonrelatives Segment

Der PC-Modus legt fest, ob ein Programmteil

- in den Speicher auf eine absolute, vom Programmierer festgelegte Adresse, geladen wird (absoluter Modus)

- oder auf relative Adressen, die sich aendern, abhaengig von der Groesse und der Anzahl der Programme (koderelativer Modus) und dem Umfang der Daten (datenrelativer Modus)

- oder auf Adressen, die gemeinsame Bereiche mit anderen Programmen darstellen (commonrelativer Modus).

Wurde nichts angegeben, ist der Modus koderelativ (standard).
Ein Programm kann Programmteile fuer verschiedene Speichersegmente enthalten.

#### Absoluter Modus

Der absolute Modus erzeugt nichtverschiebbaren Kode. Der Programmierer waehlt den absoluten Modus, wenn ein Programmblock stets auf festgelegte absolute Adressen zu laden ist, unabhaengig von sonstigen zu ladenden Segmenttypen (DSEG, CSEG, COMMON).

#### Datenrelativer Modus

Der datenrelative Modus erzeugt verschieblichen Kode fuer einen Programmteil, der veraenderlichen Kode enthalten kann und folglich in den RAM-Bereich geladen werden muss. Dies bezieht sich besonders auf Programmdatenbereiche. Symbole im datenrelativen Modus sind verschieblich.

**Koderelativer Modus**

Der Koderelative Modus erzeugt Kode fuer einen verschieblichen Programmteil, dessen Kodeinhalt sich nicht veraendert. Programmteile fuer PROM-Abspeicherung muessen im koderelativen Modus erzeugt werden.

**Common-relativer Modus**

Der Common-relative Modus erzeugt Kode, der in einem definierten Common-Bereich (gemeinsamen Datenbereich) geladen wird. Das ermoeglicht, den Programmodul in einen Speicherblock und gemeinsame Werte zu teilen.

Zum Aendern des Modus wird in einer Anweisungszeile eine PC-Modus-Pseudooperation verwendet.

| | |
|---|---|
| ASEG | absoluter Modus |
| DSEG | datenrelativer Modus |
| CSEG | koderelativer Modus (zugewiesener Modus) |
| COMMON | commonrelativer Modus |

Diese Pseudooperationen sind in Pkt. 1.4.1.3. detailiert beschrieben.
Die PC-Modus-Faehigkeit im Assembler gestattet dem Programmierer, verschiebbare Assemblerprogramme zu entwickeln.

Verschieblich bedeutet, dass der Programmodul auf jede beliebige Adresse auf verfuegbaren Speicher geladen und getestet werden kann (unter Verwendung des /P- und /D-Schalters des Programmverbinders).

### 1.3.3. Operationskodes und Pseudooperationen

Operationskodes sind mnemonische Namen fuer die Maschinenanweisungen (CPU-Befehle).
Pseudooperationen sind Vorschriften fuer den Assembler, nicht fuer den Mikroprozessor.
Operationskodes und Pseudooperationen werden (meist) in das Operationsfeld der Anweisungszeile eingetragen.
Eine Operation kann sein:

- jede beliebige Z80- oder 8080-Mnemonik
- eine Assembler-Pseudooperation
- ein Makro-Aufruf
- ein Ausdruck

Die Eintragungen im Operationsfeld werden vom Assembler in der folgenden Reihenfolge ausgewertet:

1. Makro-Aufruf
2. Operationskode / Pseudooperation
3. Ausdruck

Der Assembler vergleicht die Eintragung im Operationsfeld mit einer internen Liste von Makronamen. Wenn der Name gefunden wurde, wird die Makroexpansion durchgefuehrt und die entstehenden Anweisungen in den Modul eingefuegt (siehe dazu auch Abschnitt 1.4.2.). Ist die Eintragung kein Makro, versucht der Assembler die Eintragung als Operationskode auszuwerten.

War die Eintragung kein Operationskode, versucht der Assembler die Eintragung als Pseudooperation auszuwerten. Wenn die Eintragung auch keine Pseudooperation ist, wertet der Assembler die Eintragung als Ausdruck.

Wenn ein Ausdruck als Anweisungszeile ohne vorangestelltem Operationskode, Pseudooperation oder Makroname geschrieben wurde, meldet der Assembler keinen Fehler. Vielmehr setzt er voraus, dass ein definiertes Byte (Pseudooperation) vor den Ausdruck gehoert und uebersetzt diese Zeile.

Wegen der Auswertungsreihenfolge verhindert ein Makroname, der identisch einem Operationskode ist, desses Verwendung als Operationskode. Dieser Name wird dann ausschliesslich als Makroaufruf gebraucht. Wurde z.B. einem Makro der Name ADD zugewiesen, kann in diesem Programm ADD nicht mehr als Operationskode verwendet werden!

### 1.3.4. Argumente (Ausdruecke)

Die Argumente fuer die Operationskodes und Pseudooperationen werden gewoehnlich als Ausdruecke bezeichnet, da sie mathematischen Ausdruecken aehneln, wie z.B. 5+4*3.

Die Teile eines Ausdruckes werden Operanden (5, 4 und 3 in diesem mathematischen Ausdruck) und Operatoren (+ und * sind Beispiele) genannt. Ausdruecke koennen einen oder mehrere Operanden beinhalten.

Ein-Operand-Ausdruecke sind die am haeufigsten gebrauchten Argumente.
Wenn ein Ausdruck mehr als einen Operanden hat, sind die Operanden miteinander durch einen Operator verbunden.

Beispiel: 5+4 6-3 7*2 8/7 9>8 usw.

Im Assembler sind die Operanden numerische Werte, vertreten durch Zahlen, Zeichen, Symbole oder 8080-Operationskodes. Die Operatoren koennen arithmetische oder logische sein.

Die folgenden Abschnitte definieren die vom Assembler zugelassenen Operanden und Operatoren.

### 1.3.4.1. Operanden

Operanden koennen sein

- Zahlen
- Zeichen
- Symbole
- 8080-Operationskodes

#### Zahlen

Mit diesem Assembler kann in unterschiedlichen Zahlensystemen gearbeitet werden.
Die zugewiesene Basis fuer Zahlen ist dezimal (Standard). Diese Basis kann durch die Pseudooperation .RADIX geaendert werden. Jede Basis von 2 (binaer) bis 16 (hexadezimal) kann ausgewaehlt werden. Wenn die Grundzahl groesser als 1Ø ist, werden fuer die auf 9 folgenden Ziffern A bis F verwendet. Wenn das erste Zeichen nicht numerisch ist, muss eine Ø vorangestellt werden (z.B. ØF8).
Eine Zahl wird immer in der augenblicklich zugewiesenen Grundzahl ausgewertet, falls keine der folgenden Bezeichnungen verwendet wird:

| | |
|---|---|
| nnnnB | binaer |
| nnnnD | dezimal |
| nnnnO | oktal |
| nnnnH | hexadezimal |
| Xnnnn | hexadezimal |

Zahlen sind vorzeichenlose binaere 16-Bit-Werte (im Wertebereich 0...65535). Der Ueberlauf einer Zahl beim Ueberschreiten von 2 Byte wird ignoriert, das Ergebnis bilden die niederwertigen 16 Bits.

#### ASCII-Zeichen-Ketten

Eine Zeichenkette wird gebildet aus null oder mehr Zeichen und ist begrenzt durch Anfuehrungsstriche (") oder Apostroph (').

Wenn eine durch " begrenzte Kette als Argument auftritt, wird der Wert der Zeichen einer nach dem anderen im Speicher abgelegt.

Beispiel: DB "ABC"

speichert den Kode von A auf die erste Adresse, B auf Adresse+1 und C auf Adresse+2.

Die Begrenzer koennen als Zeichen verwendet werden, wenn sie fuer jedes gewuenschte Vorkommen zweifach erscheinen.

Beispiel: "Heute ist ein ""schoener"" Tag"

speichert die Zeichenkette

Heute ist ein "schoener" Tag

Stehen keine Zeichen zwischen den Anfuehrungszeichen, wird die Kette als Nullkette ausgewertet (leere Zeichenkette).

**Zeichenkonstanten**

Diese Ketten, Zeichenkonstanten, setzen sich aus null, einem oder zwei ASCII-Zeichen zusammen, die durch Anfuehrungszeichen (") oder Apostroph (') begrenzt werden. Soll der Begrenzer Bestandteil der Kette sein, so muss er an dieser Stelle doppelt geschrieben werden.

Die Unterschiede zwischen Zeichenkonstanten und Ketten sind:

1. Eine Zeichenkonstante besteht nur aus null, einem oder zwei Zeichen.

2. Die geschriebenen Zeichen sind dann eine Zeichenkonstante, wenn der Ausdruck mehr als einen Operanden hat. Wenn die Zeichen als einziger Operand auftreten, werden sie als Kette ausgewertet und abgespeichert.

   Beispiel: 'A'+1 ist eine Zeichenkonstante, aber
   'A' ist eine Zeichenkette.

3. Der Wert einer Zeichenkonstanten wird berechnet und das Ergebnis wird gespeichert; das niederwertige Byte in die erste Adresse und das hoeherwertige Byte in die zweite Adresse.

   Beispiel: DW 'AB'+Ø

   wertet 4142H aus und speichert 42H in die erste Adresse und 41H in die zweite Adresse.

Eine Zeichenkonstante, die nur ein Zeichen beinhaltet, bekommt den Kode des Zeichens zugeordnet. Das heisst, das hoeherwertige Byte des Wertes ist Null, das niederwertige Byte ist der Kode des Zeichens.

Beispiel: Der Wert der Zeichenkonstanten 'A' ist 41ØØH.

Eine Zeichenkonstante, die aus zwei Zeichen besteht, hat als ihren Wert den ASCII-Wert des 1. Zeichens im hoeherwertigen Byte und den ASCII-Wert des 2. Zeichens im niederwertigen Byte.

Zum Beispiel ist der Wert der Zeichenkonstanten

'AB'+Ø gleich 41H*256+42H+Ø.

**Symbole in Ausdruecken**

Ein Symbol kann als Operand in einem Ausdruck verwendet werden. Das Symbol wird ausgewertet und der Wert wird fuer das Symbol eingesetzt.

**Anwendungsvorschriften fuer das Verwenden externer Symbole in Ausdruecken**

1. Externe Symbole koennen in Ausdruecken nur mit folgenden Operatoren verwendet werden:

   \+ - * / MOD HIGH LOW

2. Wurde in einem Ausdruck ein externes Symbol verwendet, so ist das Ergebnis des Ausdrucks immer extern.

**Modusvorschriften fuer Symbole in Ausdruecken**

1. Bei allen Operationen, ausser AND, OR und XOR, koennen die Operanden jeden Modus haben.

2. Fuer die Operatoren AND, OR, XOR, SHL und SHR muessen beide Operanden absolut und intern sein.

3. Enthaelt ein Ausdruck einen absoluten Operanden und einen Operanden in einem anderen Modus, wird das Ergebnis immer den anderen, nicht absoluten Modus besitzen.

4. Bei der Subtraktion zweier Operanden verschiedener Modi ist das Ergebnis absolut. Anderenfalls ist das Ergebnis vom Typ der Operanden.

5. Wird ein datenrelatives und ein koderelatives Symbol addiert, ist das Ergebnis unbekannt. Der Assembler uebergibt den Ausdruck an den Programmverbinder als unbekannt und dieser loest den Ausdruck.

**Der laufende Speicherzuordnungszaehler**

Ein zusaetzliches Symbol fuer das Argumentfeld ist der Speicherzuordnungszaehler.

Der Speicherzuordnungszaehler ist die Adresse der naechsten zu uebersetzenden Anweisung.
Er ist oft ein passender Bezugspunkt fuer die Berechnung neuer Adressen.

Anstatt sich die laufende Programmadresse zu merken oder zu berechnen, kann der Programmierer ein Symbol verwenden, das dem Assembler den Wert des aktuellen Speicherzuordnungszaehlers mitteilt.
Das laufende Speicherzuordnungszaehlersymbol ist **$**.

**8080-Operationskodes als Operanden**

8080-Operationskodes sind nur im 8080-Modus gueltige Ein-Byte-Operanden.

Waehrend der Uebersetzung wird der hexadezimale Wert des Operationskodes berechnet und als Operand verwendet.

Bei der Verwendung von 8080-Operationskodes als Operanden muss zuerst die .8080-Pseudooperation gesetzt werden (siehe Abschnitt 1.4.1.3.).
Nur das erste Byte des errechneten hexadezimalen Wertes ist ein gueltiger Operand.

Die Verwendung von runden Klammern weist den Assembler darauf hin, dass ein Byte fuer den Operationskode zu generieren ist, wo normalerweise mehr als eins generiert wird.

Beispiel:

```
MVI  A,(JMP)
ADI  (CPI)
MVI  B,(RNZ)
CPI  (INX H)
ACI  (LXI B)
```

Der Assembler meldet einen Fehler, wenn der errechnete hexadezimale Wert mehr als 1 Byte (innerhalb der runden Klammern) ergibt, wie bei (CPI 5), (LXI B,LABEL1) oder (JMP LABEL2).
Operationskodes, die normalerweise nur 1 Byte generieren, koennen als Operand ohne einschliessende runde Klammern verwendet werden.

### 1.3.4.2. Operatoren

Der Assembler erlaubt sowohl arithmetische als auch logische Operatoren.
Operatoren, die wahre oder falsche Bedingungen melden, geben wahr zurueck, wenn das Ergebnis ungleich Ø ist und falsch, wenn das Ergebnis gleich Ø ist.
Fuer "wahr" wird 0FFFFH und fuer "falsch" 0000H uebergeben.
Die folgenden arithmetischen und logischen Operatoren sind in Ausdruecken erlaubt, wobei <exp> numerische Ausdruecke sind.

| Operator | Definition |
|---|---|
| **NUL** <exp> | - Gibt wahr zurueck, wenn das Argument (ein Parameter) null ist.<br>Der Rest der Zeile nach NUL wird als Argument zu NUL genommen.<br><br>Die Bedingung IF NUL <argument> ist falsch, wenn das erste Zeichen des Arguments alles andere als ein Semikolon oder ein <ET> ist.<br><br>Bemerkung: IFB und IFNB fuehren dieselbe Funktion aus, aber sie sind einfacher zu verwenden (siehe Pkt. 1.4.1.5. ). |
| **TYPE** <exp> | - Der TYPE-Operator gibt ein Byte zurueck, das Eigenschaften seines Arguments beschreibt:<br>- den Modus und<br>- ob es extern ist oder nicht.<br><br>Das Argument von TYPE kann jeder beliebige Ausdruck sein.<br>Wenn der Ausdruck ungueltig ist, gibt TYPE Ø zurueck.<br>Dieses Byte, das von TYPE zurueckgegeben wird, ist wie folgt aufgebaut:<br><br>Bit 1 und Ø geben den Modus an:<br>Ø Ø absolut<br>Ø 1 koderelativ<br>1 Ø datenrelativ<br>1 1 commonrelativ<br><br>Bit 7 ist das externe Bit:<br>1 Ausdruck ist external<br>Ø Ausdruck ist lokal<br><br>Bit 5 ist das Definitionsbit:<br>1 Ausdruck ist lokal definiert<br>Ø Ausdruck ist undefiniert oder extern<br><br>TYPE wird meistens innerhalb von Makros verwendet, wo der Modus eines Arguments |

| Operator | Definition |
|---|---|
| | getestet werden muss, um Entscheidungen betreffs des Programmablaufs zu treffen, z.B. wenn bedingte Assemblierung enthalten ist.<br>Beispiel:<br>MARKE MACRO X<br>LOCAL Z<br>Z SET TYPE X<br>IF Z...<br><br>TYPE testet den Modus und die Zuordnung von X. Abhaengig von der Auswertung von X wird der Kodeblock beginnend mit<br>IF Z...<br>uebersetzt oder weggelassen. |
| **LOW** <exp> | - Isolieren der niederwertigen 8 Bit (low-Teil) eines absoluten 16-Bit-Wertes. |
| **HIGH** <exp> | - Isolieren der hoeherwertigen 8 Bit (high-Teil) eines absoluten 16-Bit-Wertes. |
| <exp1> ***** <exp2> | - Multiplikation |
| <exp1> **/** <exp2> | - Division |
| <exp1> **MOD** <exp2> | - Modulo (Restdivision).<br>Division von <exp1> durch <exp2> und Zurueckgeben des Restes als Wert (Modulo). |
| <exp1> **SHR** <exp2> | - Rechtsverschiebung.<br><exp1> wird um <exp2>-Bitpositionen nach rechts verschoben. |
| <exp1> **SHL** <exp2> | - Linksverschiebung.<br><exp1> wird um <exp2>-Bitpositionen nach links verschoben. |
| **-** <exp><br>(Vorzeichen) | - zeigt an, dass das folgende <exp> negativ ist (negative ganze Zahl). |
| <exp1> **+** <exp2> | - Addition |
| <exp1> **-** <exp2> | - Subtraktion |
| <exp1> **EQ** <exp2> | - Gleichheit.<br>Gibt wahr zurueck, wenn <exp1> und <exp2> gleich sind. |
| <exp1> **NE** <exp2> | - Nicht gleich.<br>Gibt wahr zurueck, wenn <exp1> und <exp2> nicht gleich sind. |

| Operator | Definition |
|---|---|
| <exp1> **LT** <exp2> | - Kleiner als. Gibt wahr zurueck, wenn <exp1> kleiner als <exp2> ist. |
| <exp1> **LE** <exp2> | - Kleiner als oder gleich. Gibt wahr zurueck, wenn <exp1> kleiner oder gleich <exp2> ist. |
| <exp1> **GT** <exp2> | - Groesser als. Gibt wahr zurueck, wenn <exp1> groesser als <exp2> ist. |
| <exp1> **GE** <exp2> | - Groesser als oder gleich. Gibt wahr zurueck, wenn <exp1> groesser oder gleich <exp2> ist. |
| **NOT** <exp> | - Negation von <exp> |
| <exp1> **AND** <exp2> | - Logisches UND Gibt wahr zurueck, wenn <exp1> und <exp2> wahr sind; gibt falsch zurueck, wenn einer oder beide <exp> falsch sind. |
| <exp1> **OR** <exp2> | - Logisches ODER Gibt wahr zurueck, wenn einer oder beide <exp> wahr sind. Gibt falsch zurueck, wenn <exp1> und <exp2> falsch sind. |
| <exp1> **XOR** <exp2> | - Exklusives ODER Gibt wahr zurueck, wenn <exp1> oder <exp2> wahr und der andere falsch ist. Gibt falsch zurueck, wenn <exp1> und <exp2> wahr oder beide falsch sind. |

Die Reihenfolge bei der Abarbeitung dieser Operatoren ist:

```
NUL, TYPE
LOW, HIGH
*, /, MOD, SHR, SHL
Vorzeichen Minus
+, -
EQ, NE, LT, LE, GT, GE
NOT
AND
OR, XOR
```

Wenn Unterausdruecke Operatoren hoeherer Rangordnung einschliessen, dann wird dieser Ausdruck zuerst berechnet.
Die Abarbeitungsreihenfolge kann durch das Verwenden runder Klammern um den Teil des Ausdrucks, der eine hoehere Rangordnung erhalten soll, veraendert werden.

Alle Operatoren ausser +, -, * und / muessen von ihren Operanden durch wenigstens ein Leerzeichen getrennt werden. Die Byte

isolierenden Operatoren (HIGH und LOW) trennen die hoeherwertigen oder die niederwertigen 8 Bits eines 16-Bit-Wertes.

## 1.4. Assembler-Eigenschaften

Der Assembler besitzt 3 generelle Vorzuege:

- . Einzelfunktions-Pseudooperationen
- . Moeglichkeit der Makroprogrammierung
- . bedingte Assemblierung

### 1.4.1. Einzelfunktions-Pseudooperationen

Einzelfunktions-Pseudooperationen haben nur **eine** Anweisungszeile, sei weisen den Assembler auf die Ausfuehrung nur einer Funktion hin (Makros und Bedingungen haben mehr als eine Kode-Zeile; man kann sie als Block von Pseudooperationen auffassen).

Die Einzelfunktions-Pseudooperationen werden in 5 Gruppen eingeteilt:

- . Auswahl der Anweisungsliste
- . Daten- und Symboldefinition
- . Speicherzuordnungszaehler-Modus
- . Dateibezogene Pseudooperationen
- . Listensteuerung

#### 1.4.1.1. Pseudooperationen zur Auswahl der Anweisungsliste

Standardmaessig ist die 8080-Mnemonik im Assembler eingestellt.

Wenn nicht die richtige Pseudooperation zur Auswahl der Anweisungsliste angegeben wurde, gibt der Assembler einen schweren Fehler fuer jene Operationskodes zurueck, die fuer die aktuelle Anweisungsliste ungueltig sind. Das heisst, .Z80 uebersetzt nur Z80-Operationskodes und .8080 nur 8080-Operationskodes.

Deshalb muss bei einem im Z80-Kode geschriebenen Assemblerprogramm die Pseudooperation .Z80 zur Auswahl dieser Anweisungsliste geschrieben werden.

Alle in diesem Kapitel aufgefuehrten Pseudooperationen werden bei beiden Anweisungslisten verarbeitet, wenn nicht auf eine Ausnahme hingewiesen wird.

**.Z80**

_.Z80 hat keine Argumente.
_.Z80 weist den Assembler auf die Uebersetzung von Z80-Operationskodes hin.

**.8080**

_.8080 hat keine Argumente.
_.8080 weist den Assembler auf die Uebersetzung von 8080-Kode hin. Die Auswahl dieses Befehlssatzes ist Standard.

Alle Operationskodes, die der Pseudooperation zur Auswahl des Befehlssatzes folgen, werden entsprechend uebersetzt, bis die alternative Pseudooperation auftritt.
Tritt ein Operationskode auf, der nicht zum ausgewaehlten Befehlssatz gehoert, gibt der Assembler einen "U-Fehler" zurueck.

### 1.4.1.2. Pseudooperation zur Daten- und Symboldefinition

Alle Programmoperationen zur Definition von Daten und Symbolen werden fuer beide Befehlssaetze unterstuetzt. (Die einzige Ausnahme bildet SET, das fuer die Z80-Anweisungsliste nicht erlaubt ist.)

#### Definiere Byte

```
DB      <exp>[,<exp>...]
DEFB    <exp>[,<exp ...]
DEFM    <exp>[,<exp>...]
```

Bemerkung:
DB wird in den folgenden Erklaerungen stellvertretend fuer alle moeglichen Definiere-Byte-Pseudooperationen verwendet.

Die Argumente <exp> der DB's sind entweder Ausdruecke oder Zeichenketten. Ketten muessen in einfache oder doppelte Anfuehrungszeichen eingeschlossen werden.

DB wird verwendet, um einen Wert (Kette oder numerisch) in einem Speicherplatz abzulegen, beginnend ab laufenden Speicherplatzzuordnungszaehler.
Ausdruecke muessen sich auf ein Byte auswerten lassen. Wenn das hoehere Byte Ø oder 255 ergibt, ergibt sich kein Fehler. Im anderen Fall wird ein A-Fehler erzeugt.

Ketten mit 3 oder mehr Zeichen koennen in Ausdruecken nicht verwendet werden (sie muessen unmittelbar von einem Komma oder dem Ende der Zeile gefolgt werden).
Die Zeichen einer 8080- oder Z80-Kette werden in der Reihenfolge ihres Auftretens gespeichert, jedes als ein Ein-Byte-Wert, das hoechstwertige Bit auf Ø gesetzt.

```
Beispiel:               DB   'AB'
                        DB   'AB' AND ØFH
                        DB   'ABC',3
erzeugt

ØØØØ' 41 42             DB   'AB'
ØØØ2' Ø2                DB   'AB' AND ØFH
ØØØ3' 41 42 43 Ø3       DB   'ABC',3
```

**Definiere Zeichen**

**DC <string>[,<string>...]**

DC speichert die Zeichen der Kette in <string> in aufeinanderfolgende Speicherplaetze, beginnend ab laufenden Speicherzuweisungszaehler.
Wie bei DB werden die Zeichen in der Reihenfolge ihres Auftretens gespeichert, jedes als ein Ein-Byte-Wert, das hoechstwertige Bit auf Ø gesetzt. Jedoch wird beim letzten Zeichen jeder Kette das hoechstwertige Bit auf 1 gesetzt.
Es wird ein Fehler erzeugt, wenn das Argument von DC eine Nullkette ist.

Beispiel:

```
                        MARKE:  DC  "ABC",'def'
```

erzeugt

```
ØØØØ'  41 42 C3 64      MARKE:  DC  "ABC",'def'
ØØØ4'  65 E6
```

**Definiere Speicher**

**DS <exp1>[,<exp2>]**
**DEFS <exp1>[,<exp2>]**

Die Pseudooperation "Definiere Speicher" reserviert einen Speicherbereich. Der Wert von <exp> gibt die Anzahl der zu reservierenden Bytes an.
Zum Initialisieren des reservierten Bereiches setzt <exp2> den gewuenschten Wert.

Wenn <exp2> weggelassen ist, wird der reservierte Speicher nicht initialisiert. Der reservierte Speicherbereich wird nicht automatisch auf ØØH initialisiert. Eine Moeglichkeit, den reservierten Speicherbereich auf ØØH zu initialisieren, bietet der /M-Schalter waehrend der Uebersetzung (siehe Pkt. 1.5.2.4.).

Alle Namen, die in Ausdruecken verwendet werden, muessen vorher definiert sein (alle Namen muessen im Pass 1 zu diesem Zeitpunkt bekannt sein).
Ansonsten wird waehrend des Pass 1 ein V-Fehler erzeugt, ein U-Fehler kann im 2. Pass erzeugt werden; ein Phasenfehler ergibt

sich wahrscheinlich, weil die Pseudooperation "Definiere Speicher" im Pass 1 keinen Kode erzeugt hat.

Beispiel:

```
        DS  1ØØH
```

reserviert 1ØØ Byte Speicherplatz, der nicht initialisiert wird, d.h., es bleiben dort die Werte stehen, die an dieser Stelle waren, bevor das Programm geladen wurde.

Soll der Bereich auf ØØH initialisiert werden, kann waehrend der Uebersetzung der /M-Schalter verwendet werden.
Soll der reservierte Speicherbereich auf Ø2 gesetzt werden, so muss die Anweisung

```
        DS  1ØØH,2
```

geschrieben werden. Es werden 1ØØH Bytes reserviert, von denen jedes Byte mit dem Wert Ø2H initialisiert wird.

## Definiere_Wort

```
DW      <exp>[,<exp>...]
DEFW    <exp>[,<exp>...]
```

Die Pseudooperation "Definiere Wort" speichert den Wert des Ausdruckes in aufeinanderfolgende Speicherplaetze, beginnend ab laufenden Speicherzuordnungszaehler. Ausdruecke sind 2-Byte Werte (Wort).
Im Speicher steht zuerst das niederwertige Byte und dann das hoeherwertige Byte (Unterschied zu DB).

Beispiel:

```
                MARKE:  DW      1234H
erzeugt
      ØØØØ' 1234    MARKE:  DW      1234H
```

**Bemerkung:** Die Bytes werden in der Liste in Reihenfolge ihres Auftretens gezeigt, nicht in der Reihenfolge, wie sie abgespeichert sind.

## Equate

```
<name> EQU <exp>
```

EQU weist <name> den Wert des Ausdruckes <exp> zu. <name> kann ein Label, ein Symbol oder eine Variable sein und spaeter in Ausdruecken verwendet werden.
Nach <name> darf kein Doppelpunkt(e) stehen.
Wenn <exp> ein External ist, wird ein Fehler erzeugt.
Wenn <name> schon einen anderen Wert als <exp> hat, erscheint ein M-Fehler.
Wenn <name> spaeter im Programm wieder (zurueck) definiert werden soll, so ist die Pseudooperation SET oder ASET anstelle

von EQU zu verwenden.
Das ist ein Unterschied zu SET.

Beispiel: BUF EQU ØF3H

**Externes Symbol**

```
EXT             <name>[,<name>...]
EXTRN           <name>[,<name>...]
EXTERNAL        <name>[,<name>...]
BYTE EXT        <name>
BYTE EXTRN      <name>
BYTE EXTERNAL   <name>
```

Diese Pseudooperation erklaert, dass der (die) Name(n) in der Liste extern ist (sind), (d.h. in einem anderen Modul definiert).
Ist einer der in der Liste auftretenden Namen im laufenden Programm definiert, wird ein M-Fehler erzeugt.
Folgen im zu assemblierenden Programm dem Namen unmittelbar 2 Doppelkreuze (z.B. NAME##), wird dieser ebenfalls als EXTERNAL erklaert.
EXTERNAL's koennen auf ein oder zwei Byte ausgewertet werden.
Bei allen externen Namen werden nur die ersten 6 Zeichen an den Programmverbinder uebergeben.
Zusaetzliche Zeichen werden intern abgeschnitten.

Beispiel: EXTRN MARKEX

Der Assembler generiert fuer diese Anweisungszeile keinen Kode beim Uebersetzen dieses Moduls.
Wenn MARKEX als Argument in einer CALL-Anweisung verwendet wird, wird nur der Kode fuer CALL erzeugt und fuer MARKEX wird ein Ø-Wert eingetragen.
Der Programmverbinder durchsucht alle geladenen Moduln nach einer PUBLIC MARKEX-Anweisung und verwendet die Definition, die er fuer MARKEX gefunden hat in der CALL MARKEX-Anweisung.

**PUBLIC-Symbol**

```
ENTRY     <name>[,<name>...]
GLOBAL    <name>[,<name>...]
PUBLIC    <name>[,<name>...]
```

Die Pseudooperation PUBLIC erklaert jeden Namen der Liste als intern und deshalb verfuegbar fuer die Verwendung in diesem Programm und in anderen, die gemeinsam geladen und mit dem Programmverbinder gebunden sind.
Es wird ein M-Fehler erzeugt, wenn der Name ein EXTERNAL oder ein Common-Block-Name ist.
Folgen im zu assemblierenden Programm dem Namen unmittelbar 2 Doppelpunkte (z.B. NAME::), wird dieser ebenfalls als PUBLIC erklaert.
Nur die ersten 6 Zeichen des PUBLIC-Symbolnamens werden an den

Programmverbinder uebergeben.
Zusaetzliche Zeichen werden intern abgeschnitten.

Beispiel:

```
            PUBLIC     MARKEX
MARKEX:     LD         HL,BER1
```

Der Assembler uebersetzt die LD-Anweisung wie ueblich, aber er generiert fuer die PUBLIC MARKEX-Anweisung keinen Kode.
Wenn der Programmverbinder im anderen Modul die EXTERN MARKEX-Anweisung findet, weiss er, dass er suchen muss, bis er diese PUBLIC MARKEX-Anweisung gefunden hat. Dann verbindet der Programmverbinder den Adresswert der MARKEX:LD HL,BER1-Anweisung mit der (den) CALL MARKEX-Anweisung(en) im (in den) anderen Modul(n).

**SET**

```
<name>  SET   <exp>     (nicht fuer den .Z80-Modus)
<name>  DEFL  <exp>
<name>  ASET  <exp>
```

Die Pseudooperation SET weist <name> den Wert von <exp> zu. <name> kann ein Label, ein Symbol oder eine Variable sein und kann spaeter in Ausdruecken verwendet werden.

Nach <name> duerfen keine Doppelpunkte stehen. Wenn <exp> ein EXTERNAL ist, wird ein Fehler erzeugt.

Die Pseudooperation SET kann im .Z80-Modus nicht verwendet werden, weil SET ein Z80-Operationskode ist.

ASET und DEFL koennen in beiden Anweisungslisten verwendet werden.
Wenn <name> spaeter neu definiert werden soll, muss eine der SET-Pseudooperationen anstelle von EQU verwendet werden.
Zum neu Definieren von <name> kann jede beliebige SET-Pseudooperation benutzt werden, unabhaengig, mit welcher Pseudooperation die urspruengliche Definition erfolgte (das Verbot von SET im .Z80-Modus wird davon nicht beruehrt).
Das ist ein Gegensatz zu EQU.

Beispiel:

```
            MARKE   ASET  BER+1ØØØH
```

Wann immer MARKE als Ausdruck (Operand) verwendet wird, wertet der Assembler den Ausdruck BER+1ØØØH aus und setzt diesen Wert fuer MARKE. Spaeter, wenn MARKE einen anderen Wert darstellen soll, wird einfach eine MARKE ASET-Anweisung mit einem anderen Ausdruck eingetragen.

```
        MARKE  ASET   BER+1ØØØH
          .
        MARKE  ASET   3ØØØH
          .
        MARKE  DEFL   6CDEH
```

### 1.4.1.3. Pseudooperation zum Zuweisen des Speicherplatz-zuordnungszaehler-Modus

Viele Pseudooperationen beziehen sich auf den aktuellen Speicherplatzzuordnungszaehler.
Der laufende Speicherplatzzuordnungszaehler ist die Adresse des naechsten Bytes, das zu erzeugen ist.
Im Assembler erhalten Symbole und Ausdruecke ihren zugewiesenen Modus (siehe Pkt. 1.3.2.).

Jeder Modus fuer ein Speichersegment wird durch den Programmverbinder entsprechend dem Typ der uebersetzten Anweisung zugeordnet.

Es gibt 4 Modi:

- absoluter Modus
- datenrelativer Modus
- koderelativer Modus
- commonrelativer Modus.

Wenn der Speicherplatzzuordnungszaehler-Modus absolut ist, dann ist er eine absolute Adresse.

Wenn der Speicherplatzzuordnungszaehler-Modus relativ ist, dann ist er eine relative Adresse und kann einen Offset einbeziehen von der absoluten Startadresse eines relativen Segments, das vom Programmverbinder geladen werden kann.

Die Pseudooperation zur Zuweisung des Speicherplatzzuordnungszaehler-Modus wird fuer die Spezifizierung genutzt, in welchem Typ ein Programmteil zu uebersetzen ist.

#### Absolutes Segment

**ASEG**

ASEG hat niemals Operanden. ASEG erzeugt nichtverschieblichen (absoluten) Kode.
ASEG setzt den Speicherplatzzuordnungszaehler auf ein absolutes Speichersegment (aktuelle Adresse). Der Standardwert ist Ø.
Nach ASEG sollte eine ORG-Anweisung mit 1Ø3H oder hoeher geschrieben werden.

#### Kode-Segment

**CSEG**

CSEG hat niemals Operanden.
Der Kode wird im koderelativen Modus (gekennzeichnet durch "'" hinte der Adresse) uebersetzt und kann in den ROM oder PROM geladen werden.
CSEG setzt den Speicherzuordnungszaehler auf das koderelative Segment des Speichers zurueck.

Die Zuordnung ist diejenige des letzten CSEG (Standard Ø), wenn nicht danach eine ORG-Anweisung folgt, die die Zuordnung aendert.

Man muss jedoch beachten, dass die ORG-Anweisung im CSEG-Modus keine absolute Adresse setzt.
Eine ORG-Anweisung im CSEG-Modus veranlasst den Assembler, zu der letzten fuer CSEG geladenen Adresse die im Argument der ORG-Anweisung festgelegte Anzahl Bytes zu addieren.

Wenn beispielsweise ORG 5Ø angegeben ist, addiert der Assembler 5Ø Bytes zu der aktuellen CSEG-Zuordnung, und dies ist dann die neue Zuordnung von CSEG.
Die Wirkung der auf ein CSEG (oder DSEG) folgenden ORG-Anweisung ist die, dass einem Modul ein Offset gegeben werden kann. ORG setzt keine absolute Adresse fuer CSEG, sondern CSEG behaelt seine Verschieblichkeit.

Will man fuer CSEG eine absolute Adresse setzen, verwendet man den /P-Schalter im Programmverbinder.

CSEG ist der Standard-Modus des Assemblers. Der Assembler beginnt automatisch mit einem CSEG und der Speicherplatzzuordnungszaehler im koderelativen Modus weist auf den Platz Ø des koderelativen Speichersegments. Alle folgenden Anweisungen werden in das koderelative Speichersegment uebersetzt, bis eine ASEG-, DSEG- oder COMMON- Pseudooperation abgearbeitet wird.

CSEG wird folglich eingetragen, damit der Assembler zum koderelativen Modus zurueckkehrt, der Speicherplatzzuordnungszaehler wird auf den Punkt des naechsten freien Platzes im koderelativen Segment zurueckgesetzt.

**Daten-Segment**

**DSEG**

Die Pseudooperation DSEG hat keine Operanden.
DSEG spezifiziert Segmente des uebersetzten verschieblichen Kodes (gekennzeichnet durch '"' hinter der Adresse), die spaeter nur in den RAM geladen werden.

DSEG setzt den Speicherplatzzuordnungszaehler auf das datenrelative Speichersegment.
Die Zuordnung des datenrelativen Zaehlers ist die des letzten DSEG (Standard Ø), wenn nicht eine folgende ORG-Anweisung die Zuordnung aendert.
Man muss jedoch beachten, dass die ORG-Anweisung im DSEG-Modus keine absolute Adresse setzt.

Eine ORG-Anweisung im DSEG-Modus veranlasst den Assembler, zu der letzten fuer DSEG geladenen Adresse die im Argument der ORG-Anweisung festgelegte Anzahl Bytes zu addieren.
Wenn beispielsweise ORG 5Ø angegeben ist, addiert der Assembler 5Ø Bytes zu der aktuellen DSEG-Zuordnung, und dies ist dann die

neue Zuordnung von DSEG.

Die Wirkung der auf ein DSEG (oder CSEG) folgenden ORG-Anweisung ist die, dass einem Modul ein Offset gegeben werden kann. ORG setzt keine absolute Adresse fuer DSEG, sondern DSEG behaelt seine Verschieblichkeit.
Will man eine absolute Adresse fuer DSEG setzen, verwendet man den /D-Schalter im Programmverbinder.

## Common-Block (gemeinsamer Bereich)

**COMMON /[<blockname>]/**

Das Argument von COMMON ist der Name des gemeinsamen Bereiches.

COMMON legt einen gemeinsamen Datenbereich fuer alle COMMON-Bloecke an, die in dem Programm benannt sind.
Wenn <blockname> weggelassen wurde oder aus Leerzeichen besteht, wird der Block als leerer Bereich betrachtet.

COMMON-Anweisungen sind nicht ausfuehrbare Speicherzuordnungsanweisungen.
COMMON weist einem gemeinsamen Speicherbereich (gekennzeichnet durch "!" hinter der Adresse) Variable, Felder und Daten zu.
Das ermoeglicht, dass sich verschiedene Programmoduln in denselben Speicherbereich teilen.

Die der COMMON-Anweisung folgenden Anweisungen werden in den COMMON-Bereich unter <blockname> uebersetzt.
Die Laenge eines COMMON-Bereiches ergibt sich aus der Anzahl Bytes, die erforderlich sind, um die in diesem COMMON-Block definierten Variablen, Felder und Daten unterzubringen. Der COMMON-Block wird beendet, wenn eine andere Pseudooperation zur Zuweisung des Speicherplatzzuordnungszaehler-Modus auftritt.
Gemeinsame Bloecke desselben Namens koennen unterschiedliche Laenge haben.
Ist die Laenge verschieden, dann muss der Programmodul mit dem laengsten gemeinsamen Block zuerst geladen werden (d.h., er muss der erste Modulname in der Kommandozeile des Programmverbinders sein).
Siehe dazu: Beschreibung des Programmverbinders.

COMMON setzt den Speicherplatzzuordnungszaehler auf den ausgewaehlten gemeinsamen Speicherblock.

Beispiel:

```
          COMMON          /DATBIN/
BER2      EQU             1ØØH
          DB              ØFFH
          DW              1234H
          DC              'WERK'
          CSEG
```

**Set Origin**

**ORG <exp>**

Der Wert des Speicherplatzzuordnungszaehlers kann zu jeder Zeit durch die Verwendung von ORG geaendert werden. Im ASEG-Modus wird der Speicherplatzzuordnungszaehler auf den Wert von <exp> gesetzt, und der Assembler weist den erzeugten Kode, beginnend mit diesem Wert, zu.

Im CSEG-, DSEG- und COMMON- Modus wird der Speicherplatzzuordnungszaehler in diesem Segment um den Wert von <exp> erhoeht, und der Assembler weist den erzeugten Kode, beginnend mit der letzten geladenen Segmentadresse plus dem Wert von <exp>, zu.

Alle in <exp> verwendeten Namen muessen im Pass 1 bekannt sein und der Wert muss entweder absolut oder im selben Bereich, wie der Speicherplatzzuordnungszaehler sein.

Beispiel:
```
DSEG
ORG   5Ø
```

Setzt den datenrelativen Speicherplatzzuordnungszaehler auf 5Ø, relativ zum Start des datenrelativen Speichersegments. Diese Methode liefert Verschieblichkeit. Die ORG <exp>-Anweisung spezifiziert im CSEG- oder DSEG- Modus keine feste Adresse, vielmehr laedt der Programmverbinder das Segment auf eine flexible Adresse, die fuer die gemeinsam zu ladenden Moduln verwendet wird.
Andererseits wird ein Programm, das mit den Anweisungen

```
ASEG
ORG 8ØØH
```

beginnt und vollstaendig im absoluten Modus uebersetzt ist, immer beginnend ab 8ØØH geladen. Es sei denn, die ORG-Anweisung wird in der Quelldatei geaendert.
Das heisst, die dem ASEG folgende ORG <exp>-Anweisung setzt das Segment auf eine feste (im Beispiel absolute) Adresse, die durch <exp> bestimmt wird.

Das gleiche Programm - im koderelativen Modus, ohne ORG-Anweisung uebersetzt - kann auf jede beliebige Adresse geladen werden, wenn an die Kommandokette des Programmverbinders der Schalter /P:<adresse> angehaengt wird.

**Verschieben**

**.PHASE <exp>**
**.DEPHASE**

_.PHASE ermoeglicht es, Kode in einen Bereich zu laden, aber nur in einem anderen Bereich mit der durch <exp> spezifizierten Adresse abzuarbeiten.
<exp> muss ein absoluter Wert sein. .DEPHASE wird verwendet, um

anzuzeigen, dass der verschobene Kodeblock zu Ende ist.
Der Modus innerhalb eines .Phase-Blockes ist absolut. Der Kode wird in den Bereich geladen, wenn die .PHASE-Anweisung erreicht wird.
Der Kode innerhalb eines Blockes wird spaeter auf die durch <exp> fuer die Ausfuehrung spezifizierte Adresse transportiert.

Beispiel:

```
                        .PHASE      1ØØH
                MARKE:  CALL        UP1
                        JP          MARKE1
                UP1:    RET
                        .DEPHASE
                MARKE1: JP          5
```

erzeugt folgende Uebersetzung

```
                        .PHASE      1ØØH
Ø1ØØ  CD Ø1Ø6   MARKE:  CALL        UP1
Ø1Ø3  C3 ØØØ7'          JP          MARKE1
Ø1Ø6  C9        UP1:    RET
                        .DEPHASE
ØØØ7' C3 ØØØ5   MARKE1: JP          5
```

_.PHASE- .... .DEPHASE-Block sind eine Moeglichkeit, den Kodeblock ab einer spezifischen absoluten Adresse auszufuehren.

### 1.4.1.4. Dateibezogene Pseudooperationen

Die dateibezogenen Pseudooperationen

- fuegen lange Kommentare ein
- geben dem Modul einen Namen
- beenden den Modul oder
- transportieren andere Dateien in das laufende Programm

**Comment-Anweisung**

**.COMMENT <delim><text><delim>**

Das erste nichtleere Zeichen, das nach .COMMENT gefunden wird, wird als Begrenzer genommen. Der dem Begrenzer folgende <text> wird ein Kommentarblock, der fortgesetzt wird bis zum naechsten Auftreten des Begrenzers <delim>. Der Text kann mehrzeilig sein.
_.COMMENT wird fuer lange Kommentare verwendet. Es ist nicht notwendig, das Semikolon zum Anzeigen des Kommentars zu schreiben.

Waehrend der Uebersetzung wird der .COMMENT-Block ignoriert und nicht uebersetzt.

Beispiel:

```
.COMMENT * hier kann jeder beliebige mehrzeilige Text
           eingetragen sein *
; Rueckkehr zur normalen Uebersetzung
```

**Programmende**

**END [<exp>]**

Die END-Anweisung bezeichnet das Programmende und muss mit <ET> abgeschlossen sein. Wenn die END-Anweisung fehlt, wird folgende Warnung erzeugt:

"%No END statement".

<exp> kann ein Label, ein Symbol, eine Zahl oder jedes andere gueltige Argument sein, das der Programmverbinder als Startpunkt fuer das Programm laden kann.

Wenn <exp> angegeben wurde, schreibt der Programmverbinder auf die Adresse 1ØØH eine Sprunganweisung zu der in <exp> angegebenen Adresse.
Wenn <exp> nicht angegeben wurde, wird an den Programmverbinder fuer dieses Programm keine Startadresse uebergeben und die Ausfuehrung beginnt mit dem ersten geladenen Modul.

**INCLUDE-Anweisung**

**INCLUDE <dateiname>**
**$INCLUDE <dateiname>**
**MACLIB <dateiname>**

Alle drei Pseudooperationen sind sinnverwandt.
Waehrend der Assemblierung fuegen die INCLUDE-Pseudooperationen Quellkode aus einer anderen Quelldatei in die laufende Quelldatei ein.

Durch die Verwendung der INCLUDE-Pseudooperation ist es nicht erforderlich, haeufig gebrauchte Anweisungsfolgen in der laufenden Quelldatei wiederholt zu schreiben.

<dateiname> ist jede fuer das Betriebssystem gueltige Dateispezifikation und muss **in grossen Buchstaben** geschrieben werden.

Die INCLUDE-Datei wird eroeffnet und in die aktuelle Quelldatei uebersetzt, unmittelbar anschliessend an die INCLUDE-Anweisung. Wenn das Dateiende erreicht worden ist, setzt der Assembler mit der auf INCLUDE folgenden Anweisung fort.
Geschachtelte INCLUDE's sind nicht erlaubt. Fuer den Fall, dass dies auftritt, wird ein Objektkode-Syntax-Fehler "O" erzeugt.
Die im Operandenfeld spezifizierte Datei muss existieren,

ansonsten wird ein "V"-Fehler erzeugt und das INCLUDE wird ignoriert.

Im Assemblerprotokoll wird der Buchstabe "C" zwischen den uebersetzten Kode und die Quellzeile gedruckt (Siehe Pkt. 1.5.3.).

**Modulname**

**NAME ('<modulname>')**

<modulname> definiert den Namen fuer den Modul. Die runden Klammern und Apostroph um den Modulnamen sind erforderlich. Nur die ersten 6 Zeichen des Modulnamens sind signifikant.
Ein Modulname kann auch mit der Pseudooperation TITLE definiert werden.
Wenn keine der beiden Pseudooperationen NAME oder TITLE vorhanden sind, wird der Modulname aus dem Namen der Quellkodedatei erzeugt.

**RADIX (Zahlenbasis)**

**.RADIX <exp>**

<exp> in einer .RADIX-Anweisung ist immer eine dezimale numerische Konstante, ohne Ruecksicht auf die aktuelle Zahlenbasis. Die .RADIX-Anweisung erlaubt die Aenderung der Standardzahlenbasis in irgendeine beliebige zwischen 2 und 16.

_.RADIX aendert nicht die Zahlenbasis auf der Liste, sondern ermoeglicht die Eingabe numerischer Werte in der gewuenschten Basis ohne spezielle Schreibweise. (Werte in anderen Zahlenbasen erfordern spezielle Schreibweisen, erlaeutert im Abschnitt "Operanden")

Die Werte im erzeugten Kode erscheinen immer in der hexadezimalen Zahlenbasis.

Beispiel:

```
DEC:    DB      2Ø
        .RADIX  2
BIN:    DB      ØØØ1111Ø
        .RADIX  16
HEX:    DB      ØCF
        .RADIX  8
OCT:    DB      73
        .RADIX  1Ø
DECI:   DB      16
HEXA:   DB      ØCH
```

erzeugt folgende Uebersetzung:

```
ØØØØ' 14      DEC:   DB     2Ø
ØØØ2                 .RADIX 2
ØØØ1' 1E      BIN:   DB     ØØØ1111Ø
ØØ1Ø                 .RADIX 16
ØØØ2' CF      HEX:   DB     ØCF
ØØØ8                 .RADIX 8
ØØØ3' 3B      OCT:   DB     73
ØØØA                 .RADIX 1Ø
ØØØ4' 1Ø      DECI:  DB     16
ØØØ5' ØC      HEXA:  DB     ØCH
```

**Request (Anforderung)**

**.REQUEST <dateiname>[,<dateiname>...]**

.REQUEST stellt an den Programmverbinder die Forderung, die in der Liste enthaltenen Dateinamen nach undefinierten externen Symbolen zu durchsuchen (externe Symbole, fuer die zum gleichen Zeitpunkt kein korrespondierendes PUBLIC-Symbol geladen ist). Die vom Programmverbinder gefundenen undefinierten Symbole werden benoetigt, um einen oder mehrere zusaetzliche Moduln fuer einen kompletten Programmverbinderlauf zu laden.

Die Dateinamen in der Liste sollten gueltige Symbole sein und weder Dateityp noch Geraetezuweisung beinhalten.
Der Programmverbinder setzt den Standarddateityp .REL und das aktuelle Laufwerk voraus.

Beispiel:

```
            .
            .
          .REQUEST SUBR1
            .
            .
```

Der Programmverbinder durchsucht SUBR1 nach externen Symbolen, fuer die kein korrespondierendes PUBLIC-Symbol in den bereits geladenen Moduln definiert ist.

### 1.4.1.5 Pseudooperationen zur Listensteuerung

Die Pseudooperationen zur Listensteuerung fuehren 2 allgemeine Funktionen aus:

- Formatsteuerung
- Listensteuerung.

Die Pseudooperationen zur Formatsteuerung ermoeglichen dem Programmierer Seitenwechsel und Kopfzeilen einzufuegen.
Das Protokollieren gesamter Assemblerdateien oder von Teilen davon wird mit den Pseudooperationen zur Listensteuerung ein- und ausgeschaltet.

**Pseudooperationen zur Formatsteuerung**

Diese Pseudooperationen erlauben das einfuegen von Seitenwechseln und das Festlegen von Titel- und Untertitelzeilen im Assemblerprotokoll.

**Form Feed**

***EJECT [<exp>]**
**PAGE [<exp>]**
**$EJECT [<exp>]**

Die Pseudooperation Seitenwechsel veranlasst den Assembler, die Ausgabe mit einer neuen Seite zu beginnen.
Er setzt ein Seitenwechsel-Steuerzeichen an das Seitenende in die Protokolldatei.
Der Wert von <exp>, falls er angegeben wurde, bestimmt den Umfang der neuen Seite (gemessen in Zeilen pro Seite) und muss im Bereich zwischen 1Ø und 255 liegen. Die Standardzeilenzahl pro Seite ist 5Ø.
*EJECT muss in der Spalte 1 beginnen.

Beispiel: *EJECT 72

Der Assembler veranlasst den Drucker, eine neue Seite zu beginnen und jedesmal 72 Programmzeilen zu drucken (Diese Angabe ist bei einer Papierlaenge von 12" und 6 Zeilen pro Zoll zu waehlen).

**Titel**

**TITLE <text>**

TITLE spezifiziert eine Kopfzeile, die auf jede Seite als erste Zeile gesetzt wird.
Wenn mehr als eine TITLE-Anweisung angegeben wird, wird ein "Q"-Fehler erzeugt.
Wenn keine NAME-Anweisung geschrieben wurde, werden die ersten 6 Zeichen des Titels als Modulname verwendet.
(Ist weder NAME noch TITLE vorhanden, wird der Modulname aus der Quelldatei genommen.)

Beispiel: TITLE PROG1

Der Modul heisst jetzt PROG1. Er kann mit diesem Namen aufgerufen werden und PROG1 wird als Kopf auf jede Protokollseite gedruckt.

**Untertitel**

**SUBTTL <text>**

SUBTTL spezifiziert einen Untertitel, der in jeden Seitenkopf in die Zeile nach dem Titel gedruckt werden soll.
<text> wird abgebrochen, wenn 60 Zeichen erreicht wurden.
In einem Programm kann eine beliebige Anzahl von Untertiteln angegeben werden.
Wenn der Assembler eine SUBTTL-Anweisung findet, wird der alte Text durch den neuen Text ersetzt.
Um den SUBTTL fuer einen Teil der Ausgabe auszuschalten, muss ein SUBTTL mit einer leeren Kette fuer <text> geschrieben werden.

Beispiel:

```
        SUBTTL SPEZIELLE E/A-ROUTINE
        .
        .
        SUBTTL
        .
        .
```

Das erste SUBTTL veranlasst den Druck von SPEZIELLE E/A-ROUTINE im Kopf auf jeder Seite.
Das zweite SUBTTL schaltet den Untertitel aus.

**Allgemeine Pseudooperationen zur Listensteuerung**

**.LIST** Protokollieren aller Zeilen mit ihrem Kode
**.XLIST** Unterdruecken des Protokollierens

_.LIST ist die Standardbedingung.
Wenn eine Protokolldatei in der Kommandozeile spezifiziert wurde, dann wird die Datei aufgelistet.
Wenn .XLIST in der Quelldatei vorgefunden wird, werden Quell- und Objektkode nicht aufgelistet.
_.XLIST wirkt bis zum Auftreten eines .LIST.
_.XLIST setzt alle anderen Pseudooperationen zur Listensteuerung (ausser .LIST) ausser Kraft.

Beispiel:

```
        .
        .
        .XLIST      ;Protokollieren wird hier abgebrochen
        .
        .
        .LIST       ;Protokollieren wird hier fortgesetzt
        .
        .
```

**Ausgeben einer Meldung auf den Bildschirm**

**.PRINTX <delim><text><delim>**

Das erste nichtleere Zeichen nach .PRINTX ist der Begrenzer <delim>.
Der folgende Text bis zum naechsten Begrenzer bzw. bis zum Zeilenende wird waehrend der Uebersetzung auf den Bildschirm ausgegeben.

_.PRINTX ist nuetzlich, um den Prozess waehrend einer langen Uebersetzung darzustellen oder den Wert von Schaltern der bedingten Assemblierung anzuzeigen.
_.PRINTX wird in beiden Paessen ausgegeben.

Wird die Ausgabe nur in einem Pass gewuenscht, verwendet man die Pseudooperation IF1 oder IF2, um zu bestimmen, in welchem Pass die Ausgabe gewuenscht wird (siehe auch Pkt. 1.4.3.).

Beispiel: PRINTX *Uebersetzung zur Haelfte fertig*

Der Assembler gibt diese Meldung zweimal bei ihrem Auftreten (Pass 1 und Pass 2) auf den Bildschirm aus.

```
IF1
.PRINTX *PASS 1 BEENDET*; Meldung nur im Pass 1
ENDIF
IF2
.PRINTX *PASS 2 BEENDET*; Meldung nur im Pass 2
ENDIF
```

Es wird in beiden Paessen jeweils eine andere Meldung ausgegeben.

**Pseudooperationen zur bedingten Listensteuerung**

Durch Pseudooperationen ist es moeglich, das Listen von bedingt auszufuehrenden Programmteilen zu steuern, bei denen die Bedingung nicht erfuellt ist.
Diese Programmteile erzeugen keinen Objektkode in den REL-Phasen. Dazu existiert ein internes Bedingungsflag FCOND.
Hat dieses Flag den Wert ON, werden diese Programmteile gelistet, sonst (Wert OFF) wird das Listen dieser Programmteile unterdrueckt.
Zur Steuerung dieses Flags FCOND koennen die drei Pseudooperationen

**.SFCOND**
**.LFCOND**
**.TFCOND**

benutzt werden.
Siehe dazu auch die Beschreibung zum /X-Schalter im Abschnitt "Schalter" (Pkt. 1.5.2.4.).

## Unterdruecken falscher Bedingungen

**.SFCOND**

Die Pseudooperation .SFCOND setzt das FCOND-Flag unbedingt ON.

## Protokollieren

**.LFCOND**

Die Pseudooperation .LFCOND setzt das FCOND-Flag unbedingt OFF.

## Umkippen der Listensteuerung falscher Bedingungen

**.TFCOND**

Die Pseudooperation .TFCOND setzt das FCOND-Flag bedingt in Abhaengigkeit des Standardbedingungsflags STFCOND.
_.TFCOND triggert den Wert des STFCOND-Flags und weist dem FCOND-Flag den neuen STFCOND-Wert zu.

Die Initialbelegung des FCOND-Flags bei Programmbeginn erfolgt durch Zuweisung des STFCOND-Wertes.
Der Initialwert des STFCOND-Flags ist vom Aufruf des ASM-Programmes abhaengig. Normalerweise ist dieser Wert ON. Bei Setzen des Schalters /X wird dieser Wert OFF.

Die folgende Uebersicht gibt einen Zusammenhang der Pseudooperationen und des FCOND-Flags in Abhaengigkeit des Schalters /X.

| Pseudooperation | kein /X | mit /X |
|---|---|---|
| ------- | ON | OFF |
| .SFCOND | OFF | OFF |
| .LFCOND | ON | ON |
| .TFCOND | OFF | ON |
| .TFCOND | ON | OFF |
| .SFCOND | OFF | OFF |
| .TFCOND | OFF | ON |
| .TFCOND | ON | OFF |
| .TFCOND | OFF | ON |

## Pseudooperationen zur Listensteuerung der Makro-Erweiterung

Pseudooperationen zur Listensteuerung der Makro-Erweiterung steuern die Protokollierung der Zeilen innerhalb eines Makros oder einer Wiederholungs-Pseudooperation (REPT, IRP, IRPC) und koennen nur innerhalb eines Makros oder eines Wiederholungs-blockes verwendet werden.

### Ausschliessen von Makrozeilen, die keinen Kode erzeugen

**.XALL**

_.XALL ist Standard. .XALL protokolliert Quell- und Objektkode, der durch einen Makro erzeugt wird.
Quellzeilen, die keinen Kode erzeugen, werden nicht protokolliert.

### Protokollieren Makrotext

**.LALL**

_.LALL protokolliert den kompletten Text fuer die ganze Makroerweiterung einschliesslich der Zeilen, die keinen Kode erzeugen.

### Unterdruecken der Makro-Protokollierung

**.SALL**

_.SALL unterdrueckt das Protokollieren des gesamten Textes und des durch den Makro erzeugten Objektkodes.

Zu diesen 3 Pseudooperationen das nachfolgende Beispiel:
Der Makro

```
MAC  MACRO     X,Y
Y1   DEFL      Y
;Kommentarzeile
     LD        X,Y1
ENDM
```

erzeugt bei Aufruf durch MAC B,4 bei jeweiliger Verwendung der Pseudooperation folgende Uebersetzung:

```
                    .XALL
                    MAC  B,4
ØØØØ'  Ø6 Ø4    +       LD        B,Y1
                    .LALL
                    MAC  B,4
ØØØ4            +   Y1   DEFL      4
                +   ;Kommentarzeile
ØØØ2'  Ø6 Ø4    +       LD        B,Y1
                    .SALL
                +   MAC  B,4
```

..

**Pseudooperationen zur Steuerung der Cross-Referenz**

Es ist moeglich, die Cross-Referenz nur fuer einen Teil, aber nicht fuer das ganze Programm zu generieren. Dazu verwendet man die Pseudooperation .CREF und .XCREF in der Quelldatei fuer den Assembler.

Diese beiden Pseudooperationen koennen an jede beliebige Stelle im Programm in das Operationsfeld geschrieben werden.
Wie alle Pseudooperationen zur Listensteuerung haben sie keine Argumente.

**Unterdruecken der Cross-Referenz**

**.XCREF**

_.XCREF schaltet .CREF (Standard) aus. .XCREF bleibt wirksam bis der Assembler ein .CREF trifft.
_.XCREF wird verwendet, um in einem ausgewaehlten Teil der Dateien das Erzeugen einer Cross-Referenz zu unterdruecken.
Weil weder .CREF noch .XCREF Wirkung haben, wenn nicht der /C-Schalter in der Assembler-Kommandozeile gesetzt ist, muss _.XCREF nicht verwendet werden, wenn eine normale Liste (ohne Cross-Referenz) gewuenscht wird. Das wird auch erreicht, wenn /C in der Kommandozeile weggelassen wird.

**Erzeugen einer Cross-Referenz**

**.CREF**

_.CREF ist die Standard-Bedingung.
_.CREF wird verwendet, um mit der Erstellung der Cross-Referenz fortzufahren, nachdem diese durch .XCREF gestoppt worden war.
_.CREF bleibt wirksam bis ein .XCREF auftritt.
_.CREF hat nur dann Wirkung, wenn der /C-Schalter in der Kommandozeile des Assemblers gesetzt ist.

## 1.4.2. Makrofaehigkeit

Die Makrofaehigkeit erlaubt, Bloecke von Anweisungen zu schreiben, die wiederholt verwendet werden koennen, ohne dass sie wieder aufgeschrieben werden muessen.
Diese Anweisungsbloecke beginnen entweder mit der Pseudooperation Makrodefinition oder mit einer der Wiederholungspseudooperationen. Sie enden mit ENDM. Alle Makro-Pseudooperationen koennen innerhalb eines Makroblockes verwendet werden.
Die Schachtelungstiefe von Makros ist nur durch den Speicherplatz begrenzt.
Die Makrofaehigkeit des Assemblers beinhaltet folgende Pseudooperationen, die nachfolgend erlaeutert werden:

- Makrodefinition

```
MACRO
```

- Wiederholungen

```
REPT    (Wiederholung)
IRP     (undefinierte Wiederholung)
IRPC    (undefinierte Wiederholung mit Zeichen)
```

- Abschluss

```
ENDM
EXITM
```

- nur einmal vorkommende Symbole innerhalb des Makroblockes

```
LOCAL
```

Die Makrofaehigkeit unterstuetzen ausserdem einige spezielle Makrooperatoren:

```
&      ;;      !      %
```

### 1.4.2.1. Pseudooperation zur Makrodefinition

```
<name> MACRO <dummy>[,<dummy>...]
       .
       .
       .
       ENDM
```

Der Anweisungsblock von MACRO bis ENDM bildet den Kern des Makros oder die Makrodefinition.
Dabei ist <name> ein LABEL und unterliegt den Regeln fuer die Bildung von Symbolen. <name> kann jede beliebige Laenge haben, aber nur die ersten 16 Zeichen werden an den Programmverbinder uebergeben.
Nachdem der Makro definiert worden ist, kann <name> zum Aufrufen des Makros verwendet werden.
<dummy> ist ein symbolischer Parameter, der durch den echten Parameter durch Eins-zu-Eins-Textsubstitution bei der Verwendung des Makros ersetzt wird.
Jedes <dummy> kann 32 Zeichen lang sein. Die Anzahl der symbolischen Parameter ist nur durch die Laenge der Zeile begrenzt. Die symbolischen Parameter werden durch Kommas voneinander getrennt.
Der Assembler interpretiert alle Zeichen zwischen zwei Kommas als einen einzigen symbolischen Parameter.

Bemerkung:
Ein symbolischer Parameter wird ausschliesslich als solcher erkannt.

Wenn zum Beispiel ein Registername (A oder B) als symbolischer Parameter verwendet wurde, wird er waehrend der Erweiterung durch einen Parameter ersetzt.

Ein Makroblock wird nicht bei seinem Auftreten uebersetzt, sondern dann, wenn er aufgerufen wird, erweitert der Assembler die Makro-Aufruf-Anweisung durch den passenden Makroblock.
Wenn die Pseudooperation TITLE, SUBTTL oder NAME fuer den Teil des Programms, wo der Makroblock erscheint, verwendet werden soll, ist Vorsicht beim Schreiben dieser Anweisungszeile geboten.

Beispiel: SUBTTL MACRO DEFINITION

Der Assembler uebersetzt die Anweisung als Makrodefinition mit dem Namen SUBTTL und DEFINITION als symbolischen Parameter. Um das zu vermeiden, koennte das Wort MACRO abgeaendert werden.

**Makroaufruf**

Soll ein Makro verwendet werden, muss eine Makroaufruf-Anweisung geschrieben werden.

**<name> <parameter>[,<parameter>...]**

<name> ist gleich <name> des Makroblockes.
<parameter> ersetzt Eins-zu-Eins <dummy>. Die Anzahl der Parameter wird nur durch die Laenge der Zeile begrenzt. Die Parameter muessen durch Kommas voneinander getrennt werden.
Wenn um Parameter, die durch Kommas getrennt sind, spitze Klammern geschrieben sind, uebergibt der Assembler alles, was in den spitzen Klammern steht als einen einzigen Parameter.

Beispiel: MAC 1,2,3,4,5

uebergibt 5 Parameter an den Makro, aber

MAC <1,2,3,4,5>

uebergibt nur einen Parameter.

Die Anzahl der Parameter in der Makroaufruf-Anweisung muss nicht mit der Anzahl der symbolischen Parameter (<dummy>) in der Makrodefinition uebereinstimmen.
Wenn der Aufruf mehr Parameter enthaelt als die Definition, werden die uebrigen ignoriert. Wenn es weniger sind, werden die uebriggebliebenen symbolischen Parameter Null gesetzt.
Nach jeder Makroaufruf-Anweisung wird der Makroblock in den uebersetzten Kode eingefuegt.

Beispiel:

```
               EXCHNG    MACRO     X,Y
                         PUSH      X
                         PUSH      Y
                         POP       X
                         POP       Y
                         ENDM
```

Im folgenden Programm wird ein Makroaufruf verwendet:

```
                         LD        A,(2FH)
                         LD        H,A
                         LD        A,(3FH)
                         LD        D,A
                         EXCHNG    HL,DE
```

Durch ASM wird folgende Uebersetzung erzeugt:

```
ØØØØ'  3A ØØ2F           LD        A,(2FH)
ØØØ3'  67                LD        H,A
ØØØ4'  3A ØØ3F           LD        A,(3FH)
ØØØ7'  57                LD        D,A
                         EXCHNG    HL,DE
ØØØ8'  E5         +      PUSH      HL
ØØØ9'  D5         +      PUSH      DE
ØØØA'  E1         +      POP       HL
ØØØB'  D1         +      POP       DE
```

### 1.4.2.2. Wiederholungs-Pseudooperationen

Die Pseudooperationen ermoeglichen es, Operationen in einem Kodeblock so oft zu wiederholen, wie spezifiziert wurde.
Die Hauptunterschiede zwischen den Wiederholungs- und den Makropseudooperationen sind:

1. MACRO gibt dem Block einen Namen, mit welchem er in den Kode an die Stelle, wo er gebraucht wird, aufgerufen werden kann. Der Makro kann in mehreren verschiedenen Programmen durch einfaches Schreiben einer Makroaufruf-Anweisung verwendet werden.

2. MACRO ermoeglicht das Uebergeben von Parametern an den Makroblock, wenn er aufgerufen wird; diese Parameter koennen sich aendern.

Die Wiederholungs-Pseudooperationen muessen als ein Teil des Kodeblockes zugewiesen sein.
Das Verwenden der Wiederholungs-Pseudooperationen ist bequem, wenn die Parameter im Voraus bekannt sind und nicht geaendert werden und wenn die Wiederholung bei jeder Programmausfuehrung realisiert werden soll.
Mit der Makro-Pseudooperation muss der Makro jedesmal, wenn er gebraucht wird, aufgerufen werden!
Beachte, dass jeder Wiederholungsblock mit der ENDM-Pseudooperation abgeschlossen werden muss.

**Wiederholung**

```
REPT <exp>
.
.
ENDM
```

Wiederholen des Anweisungsblockes zwischen REPT und ENDM <exp>-mal, wobei <exp> eine vorzeichenlose 16-bit-Zahl ist.

Wenn <exp> ein externes Symbol oder undefinierte Operanden enthaelt, wird ein Fehler generiert.

Beispiel:

```
          X    DEFL    Ø
               REPT    1Ø
          X    DEFL    X+1
               DB      X
               ENDM
```

erzeugt folgendes Uebersetzungsprotokoll:

```
ØØØØ           X     DEFL    Ø
                     REPT    1Ø
               X     DEFL    X+1
                     REPT    1Ø
                     ENDM
ØØØØ'  Ø1      +     DB   X
ØØØ1'  Ø2      +     DB   X
ØØØ2'  Ø3      +     DB   X
ØØØ3'  Ø4      +     DB   X
ØØØ4'  Ø5      +     DB   X
ØØØ5'  Ø6      +     DB   X
ØØØ6'  Ø7      +     DB   X
ØØØ7'  Ø8      +     DB   X
ØØØ8'  Ø9      +     DB   X
ØØØ9'  ØA      +     DB   X
```

**Undefinierte_Wiederholung**

```
IRP <dummy>,<parameter>
 .
 .
ENDM
```

Die <parameter> **muessen** in spitze Klammern eingeschlossen werden!

Die Parameter koennen gueltige Symbole, Zeichenketten, Zahlen oder Zeichenkettenkonstanten sein.
Der Anweisungsblock wird fuer jeden Parameter wiederholt. Jede Wiederholung ersetzt den naechsten Parameter fuer jedes Auftreten von <dummy> in dem Block.
Wenn ein Parameter Null ist (z.B. <>), wird der Block einmal

mit einem Nullparameter ausgefuehrt.

Beispiel:
```
IRP  X,<1,2,3,4,5,6,7,8,9,1Ø>
DB   X
ENDM
```

Dieses Beispiel erzeugt denselben Kode (DB 1 - DB 1Ø) wie im Beispiel bei REPT.
Wenn IRP innerhalb eines Makrodefinitionsblockes verwendet wird, werden die spitzen Klammern um die Parameter in der Makroaufruf-Anweisung entfernt, bevor die Parameter an den Makroblock uebergeben werden.

Ein Beispiel, das den gleichen Kode wie oben erzeugt, verdeutlicht das Entfernen der Klammern von den Parametern:

```
MAC     MACRO    X
           IRP     Y,<X>
           DB      Y
           ENDM
        ENDM
```

Wenn die Makroaufruf-Anweisung

```
MAC <1,2,3,4,5,6,7,8,9,1Ø>
```

uebersetzt wird, dann sieht die Makroerweiterung folgendermassen aus:
```
IRP       Y,<1,2,3,4,5,6,7,8,9,1Ø>
DB        Y
ENDM
```

Die spitzen Klammern um die Parameter sind entfernt worden und der Inhalt wird als ein einziger Parameter uebergeben.

## Undefinierte Wiederholung mit Zeichen

```
IRPC <dummy>,<string>
.
.
ENDM
```

Die Anweisungen in dem Block werden fuer jedes Zeichen der Zeichenkette wiederholt.
Jede Wiederholung ersetzt das naechste Zeichen der Zeichenkette fuer jedes Auftreten von <dummy> im Block.

Beispiel:
```
IRPC      X,Ø123456789
DB        X+1
ENDM
```

Dieses Beispiel erzeugt den gleichen Kode (DB 1 - DB 1Ø) wie die beiden vorhergehenden Beispiele.

### 1.4.2.3. Beendigungs-Pseudooperationen

**Makro-Ende**

**ENDM**

ENDM teilt dem Assembler mit, dass der Makro- oder Wiederholungsblock zu Ende ist.
Jede Pseudooperation MACRO, REPT, IRP, IRPC muss mit einem zugehoerigen ENDM beendet werden. Ist dies nicht der Fall, so wird am Ende jedes Passes durch ASM die Meldung
"Unterminated REPT/IRP/IRPC/MACRO" (nicht beendetes REPT/...) erzeugt.

Ein ueberzaehliges ENDM verursacht einen "O"-Fehler.
Wenn ein Makro- oder Wiederholungsblock verlassen werden soll, bevor die Erweiterung vollstaendig ist, wird die Pseudooperation EXITM verwendet.

**Verlassen des Makro**

**EXITM**

EXITM wird innerhalb eines Makro- oder Wiederholungsblockes zum vorzeitigen Beenden einer Erweiterung verwendet, wenn irgendeine Bedingung die weitere Erweiterung unnoetig oder unerwuenscht macht. Meistens wird EXITM in Verbindung mit einer bedingten Pseudooperation verwendet.

Wenn ein EXITM uebersetzt wird, wird die Erweiterung sofort abgebrochen. Die restliche Erweiterung oder uebrige Wiederholungen werden nicht generiert.

Wenn der Block, der das EXITM enthaelt, in einem anderen Block verschachtelt ist, wird die Erweiterung in der aeusseren Ebene fortgesetzt.

Beispiel:

```
MAC       MACRO     X
Y         DEFL      Ø
               REPT      X
     Y         DEFL      Y + 1
               IFE       Y - ØFFH ; Test Y
               EXITM               ; wenn Ø, verlassen REPT
               ENDIF
               DB        Y
               ENDM
          ENDM
```

### 1.4.2.4. Pseudooperation fuer Makro-Symbole

**LOCAL <dummy>[,<dummy>...]**

Die Pseudooperation LOCAL wird nur innerhalb eines Makrodefinitionsblockes verwendet.

Wenn LOCAL ausgefuehrt wird, legt der Makroassembler ein einmaliges Symbol fuer jedes <dummy> an und ersetzt dieses Symbol fuer jedes Auftreten von <dummy> in der Erweiterung.
Diese einmaligen Symbole werden meistens zur Definition einer Marke innerhalb eines Makros verwendet, auf diese Weise vermeidet man Mehrfachdefinitionen bei mehrfacher Erweiterung des Makros.

Die vom Assembler angelegten Symbole reichen von ..ØØØ1 bis _..FFFF. Der Anwender sollte eigene Symbole der Form ..nnnn vermeiden! Die LOCAL-Anweisung muss allen anderen Anweisungstypen in der Makrodefinition vorausgehen.

Beispiel:

```
             MAC       MACRO     NUM,Y
                       LOCAL     A,B,C,D,E
             A:        DB        7
             B:        DB        8
             C:        DB        Y
             D:        DB        Y+1
             E:        DW        NUM+1
                       JP        A
                       ENDM

Der Aufruf             MAC       ØCØØH,ØBEH
```

erzeugt folgende Uebersetzungsliste:

```
ØØØØ'  Ø7          +    ..ØØØØ:   DB        7
ØØØ1'  Ø8          +    ..ØØØ1:   DB        8
ØØØ2'  BE          +    ..ØØØ2:   DB        ØBEH
ØØØ3'  BF          +    ..ØØØ3:   DB        ØBEH+1
ØØØ4'  ØCØ1        +    ..ØØØ4:   DW        ØCØØH+1
ØØØ6'  C3 ØØØØ'    +              JP        ..ØØØØ
```

### 1.4.2.5. Spezielle Makro-Operatoren

Besondere spezielle Operatoren koennen in einem Makroblock verwendet werden, um zusaetzliche Assemblerfunktionen auszuwaehlen.

**&** Ampersand verkettet Text oder Symbole. (Das Ampersand kann nicht in einer Makroaufruf-Anweisung verwendet werden.)
Ein symbolischer Parameter wird in der Erweiterung nicht ersetzt, wenn nicht unmittelbar vorher ein & steht.
Ein Symbol wird aus Text und symbolischem Parameter gebildet, indem zwischen beide ein & geschrieben wird.

Beispiel:

```
                ERRGEN    MACRO     X
                           ERROR&X:  PUSH      BC
                                     LD        B,'&X'
                                     JP        ERROR&X
                                     ENDM
```

Der Aufruf ERRGEN A erzeugt:

```
ØØØØ'  C5            +    ERROR&A:  PUSH      BC
ØØØ1'  Ø6 41         +              LD        B,'A'
ØØØ3'  C3 ØØØØ'      +              JP        ERROR&A
```

**;;** In einem Block wird ein Kommentar, dem 2 Semikolons vorausgehen, nicht als Teil der Erweiterung erhalten (er erscheint bei .LALL nicht auf dem Protokoll).
Ein Kommentar nach nur einem Semikolon wird erhalten und erscheint in der Erweiterung.

**!** Ein Ausrufezeichen kann in einem Argument geschrieben werden, um anzuzeigen, dass das naechste Zeichen als Literal genommen werden soll.
Deswegen ist !; dasselbe wie <;>.

**%** Das Prozentzeichen wird nur in einem Makroargument zum Konvertieren des auf das % folgenden Ausdruckes (meistens ein Symbol) in eine Zahl in der aktuellen Zahlenbasis (festgesetzt mit .RADIX) verwendet.
Waehrend der Makroerweiterung ersetzt diese Zahl den symbolischen Parameter.
Die Verwendung des %-Operators erlaubt den Makroaufruf mit einer Zahl.
(Meistens ist ein Makroaufruf ein Aufruf mit einer Referenz, wobei der Text des Makroarguments exakt den symbolischen Parameter ersetzt.)
Der auf % folgende Ausdruck muss den gleichen Regeln genuegen, wie der Ausdruck bei einer DS-Pseudooperation. Das heisst, es ist ein gueltiger Ausdruck erforderlich, der eine absolute Konstante (nichtverschieblich) ergibt.

Beispiel:

```
                PRINTE    MACRO     MSG,N
                .PRINTX     *MSG,N*
                ENDM

                SYM1      EQU       1ØØ
                SYM2      EQU       2ØØ
                          PRINTE <SYM1+SYM2=>,%(SYM1+SYM2)
```

Normalerweise wuerde beim Makroaufruf der symbolische Parameter N durch die Kette (SYM1+SYM2) ersetzt.
Das Resultat waere:

```
PRINTX*SYM1+SYM2=,(SYM1+SYM2)*
```

Wenn das % vor dem Parameter steht, wird folgendes erzeugt:

```
PRINTX*SYM1+SYM2=,3ØØ*
```

..

## 1.4.3. Pseudooperationen zur bedingten Assemblierung

Die bedingten Pseudooperationen ermoeglichen dem Anwender Kodebloecke zu entwerfen, die spezielle Bedingungen testen und entsprechend vorgehen.

Alle Bedingungen haben folgendes Format:

```
IFxxxx [argument]          |          COND [argument]
 .                         |           .
 .                         |           .
 .                         |           .
[ELSE                      |          [ELSE
 .                         |           .
 .                         |           .
 .  ]                      |           .  ]
ENDIF                      |          ENDC
```

Zu jedem IFxxxx muss ein zugehoeriges ENDIF die Bedingung abschliessen.
Zu jedem COND muss ein zugehoeriges ENDC die Bedingung abschliessen. Andererseits wird die Meldung "Unterminated Conditional" (nicht beendete Bedingung) am Ende jedes Passes erzeugt.
Ein ENDIF ohne zugehoeriges IF oder ein ENDC ohne zugehoeriges COND verursacht einen "C"-Fehler.

Der Assembler bewertet eine Bedingungsanweisung mit "wahr" (ist gleich FFFFH oder -1 oder jeder Wert ungleich Null) oder mit "falsch" (ist gleich Null).

Der Kode in dem Bedingungsblock wird uebersetzt, wenn der Wert der Bedingung entspricht, die in der Bedingungsanweisung definiert wurde. Wenn der Wert dem nicht entspricht, ignoriert der Assembler den Bedingungsblock entweder vollstaendig oder, wenn er eine optionale ELSE-Anweisung enthaelt, uebersetzt er nur den ELSE-Teil.

Bedingungen koennen bis zu 255mal geschachtelt werden. Jedes Argument einer Bedingung muss im Pass 1 bekannt sein, um einen "V"-Fehler und unkorrekte Auswertung zu vermeiden. Der Ausdruck fuer IF/IFT/COND und IFF/IFE muss einen Wert haben, der vorher definiert wurde und der absolut ist.

Wenn der Name nach einem IFDEF oder IFNDEF definiert wurde, betrachtet der Assembler ihn im Pass 1 als undefiniert; aber er wird im Pass 2 definiert.

Jeder Bedingungsblock kann optional die Pseudooperation ELSE enthalten, die die Moeglichkeit gibt, alternativen Kode zu erzwingen, wenn die entgegengesetzte Bedingung auftritt.
Fuer ein IFxxxx/COND ist nur ein ELSE erlaubt.
Das ELSE ist immer mit dem zuletzt eroeffneten IF verbunden.
Eine Bedingung mit mehr als einem ELSE oder ein ELSE ohne eine Bedingung wird einen "C"-Fehler verursachen.

**Bedingte Pseudooperation**

**IF <exp>**
**IFT <exp>**
**COND <exp>**
Die Anweisungen innerhalb des Bedingungsblockes werden uebersetzt, wenn <exp> nicht Null ergibt.

**IFE <exp>**
**IFF <exp>**
Die Anweisungen innerhalb des Bedingungsblockes werden uebersetzt, wenn <exp> den Wert Null hat.

**IF1**
Die Anweisungen innerhalb des Bedingungsblockes werden uebersetzt, wenn sich der Assembler im Pass 1 befindet.

**IF2**
Die Anweisungen innerhalb des Bedingungsblockes werden uebersetzt, wenn sich der Assembler im Pass 2 befindet.

**IFDEF <symbol>**
Die Anweisungen innerhalb des Bedingungsblockes werden uebersetzt, wenn das Symbol definiert oder als EXTERNAL erklaert wurde.

**IFNDEF <symbol>**
Die Anweisungen innerhalb des Bedingungsblockes werden uebersetzt, wenn das Symbol nicht definiert und nicht als EXTERNAL erklaert wurde.

**IFB <arg>**
Die spitzen Klammern um <arg> sind erforderlich. Die Anweisungen im Bedingungsblock werden uebersetzt, wenn das Argument leer (nicht angegeben) oder Null (<>) ist.

**IFNB <arg>**
Die spitzen Klammern um das <arg> sind erforderlich. Die Anweisungen innerhalb des Bedingungsblockes werden uebersetzt, wenn das Argument nicht leer ist.
Dies wird verwendet, um symbolische Parameter zu testen.

**IFIDN <<arg1>>,<<arg2>>**
Die spitzen Klammern um <arg1> und <arg2> sind erforderlich.
Die Anweisungen innerhalb des Bedingungsblockes werden uebersetzt, wenn die Kette <arg1> identisch der Kette <arg2> ist.

**IFDIF <<arg1>>,<<arg2>>**
Die spitzen Klamern um <arg1> und <arg2> sind erforderlich.
Die Anweisungen innerhalb des Bedingungsblockes werden uebersetzt, wenn die Kette <arg1> verschieden von der Kette <arg2> ist.

**ELSE**
ELSE bietet die Moeglichkeit, alternativen Kode zu erzeugen, wenn die entgegengesetzte Bedingung auftritt. ELSE kann mit jeder

bedingten Pseudooperation verwendet werden.

..

**ENDIF**
**ENDC**

Diese Pseudooperationen schliessen den Bedingungsblock ab. Sie muessen zu jeder verwendeten bedingten Pseudooperation geschrieben werden.
ENDIF muss ein zugehoeriges IFxxxx haben und ENDC ein zugehoeriges COND.

## 1.5. Abarbeitung des Assemblers

Wenn das Assemblerprogramm vollstaendig entworfen ist, kann es uebersetzt werden.
Der Assembler uebersetzt die Quelldateianweisungen einschliesslich der Erweiterung von Makros und Wiederholungspseudooperationen.

Das Ergebnis ist ein verschieblicher Objektkode, der mit dem Programmverbinder gebunden und geladen werden kann. Der verschiebliche Objektkode kann in eine Diskettendatei, die vom Assembler den Dateityp .REL bekommt, geschrieben werden. Die uebersetzte Datei (.REL) ist nicht ausfuehrbar.
Die Datei wird erst nach der Behandlung durch den Programmverbinder ausfuehrbar.

Der Assembler benoetigt ungefaehr 19K Speicherplatz und hat eine Abarbeitungsgeschwindigkeit von mehr als 1ØØØ Zeichen je Minute.

Der Assembler uebersetzt die Quelldatei in zwei Paessen.
Waehrend des 1. Passes wertet er die Programmanweisungen aus, berechnet, wieviel Kode er erzeugen wird, bildet eine Symboltabelle, in der jedem Symbol ein Wert zugewiesen wird und erweitert die Makroaufruf-Anweisungen.

Waehrend des 2. Passes setzt er in die Symbole und Ausdruecke die Werte aus der Symboltabelle ein, erweitert erneut die Makroaufruf-Anweisungen und gibt den verschieblichen Kode aus. Der Assembler prueft die Werte der Symbole, Ausdruecke und Makros waehrend beider Paesse. Er gibt einen Phasenfehlerkode zurueck, wenn der Wert waehrend des 2. Passes nicht mit dem Wert waehrend des 1. Passes uebereinstimmt.

Bevor mit dem Assembler gearbeitet werden kann, muss die Diskette mit dem Programm ASM.COM in das zu verwendende Laufwerk eingelegt werden.
Die Diskette, auf der sich die Quelldatei befindet, muss ebenfalls in einem Laufwerk liegen.

..
..

### 1.5.1 Aufrufen des Assemblers

Der Assembler wird aufgerufen durch die Eingabe von:

**ASM**

Die Programmdatei ASM.COM wird geladen. Der Assembler gibt einen Stern ("*") aus und zeigt damit an, dass er bereit ist, eine Kommandozeile entgegenzunehmen.

### 1.5.2. Die Kommandozeile des Assemblers

Die Kommandozeile des Assemblers setzt sich aus 4 Feldern zusammen, die wie folgt aussehen kann:

**[Objekt][,[List]]=Source[/Switch]**

Die Kommandozeile kann als eigene Zeile oder auf dieselbe Zeile wie das ASM-Kommando geschrieben werden.
(Wenn Kommandozeile und ASM-Kommando auf derselben Zeile stehen, kehrt der Assembler nicht mit * zurueck, sondern gibt die Steuerung nach Beenden an das Betriebssystem.)
Bei getrennter Eingabe von ASM und Kommandozeile besteht zusaetzlich die Moeglichkeit, eine weitere Uebersetzung auszufuehren, ohne dass der Assembler wieder aufgerufen wird.

Wenn die Uebersetzung beendet ist, meldet sich der Assembler dann sofort mit einem * und wartet auf eine neue Kommandozeile. Der Assembler wird in diesem Fall durch Druecken von ^C beendet.

Wenn nur ein Programm uebersetzt werden soll, ist es bequemer, das ASM-Kommando und die Kommandozeile auf dieselbe Zeile zu schreiben; es erfordert weniger Eingaben und bietet die Moeglichkeit, die Assembler-Operation einem SUBM-Kommando mitzuteilen.

Wenn das ASM-Kommando und die Kommandozeile zusammenstehen, kehrt der Assembler automatisch nach Abschluss zum Betriebssystem zurueck.

Bemerkung:
Wenn die Kommandozeile eine vom ASM-Kommando separate Zeile ist, kann die Kommandozeile nur in Grossbuchstaben eingegeben werden!
Wird anders gearbeitet, erscheint die Meldung "?Command Error".
Wenn das ASM-Kommando und die Kommandozeile eine Zeile bilden, koennen die Eingaben in grossen oder kleinen Buchstaben oder gemischt gemacht werden. Das Betriebssystem wandelt vor der Uebergabe alle Eingaben in grosse Buchstaben um.

### 1.5.2.1. Source (=dateiname)

Um ein Quellprogramm zu uebersetzen, muss wenigstens ein Gleichheitszeichen (=) und der Quelldateiname eingegeben werden.

Das **"=dateiname"** zeigt an, welche Quelldatei zu uebersetzen ist. Wenn die Diskette mit der Quelldatei nicht im aktuell zugewiesenen Laufwerk liegt, muss die Laufwerksauswahl als Teil von dateiname mit spezifiziert werden.
Wird im Dateinamen kein Dateityp mit angegeben, nimmt der Assembler an, dass der Dateityp .MAC ist. Ist er nicht .MAC, so muss er als Teil von dateiname unbedingt angegeben werden.

Ueber andere Moeglichkeiten der Zuweisung von Geraet und Dateityp wird im Pkt. 1.5.2.5. geschrieben.

Die Eintragung **source** ist die einzige, die fuer ASM unbedingt erforderlich ist.

Das einfachste Kommando lautet:

**ASM =source**

Dieses Kommando veranlasst den Assembler, die Quelldatei zu uebersetzen und das Ergebnis in einer verschieblichen Objektkodedatei (genannt .REL-datei) mit demselben Namen wie die Quelldatei abzulegen.

Beispiel:

Wenn die Quelldatei PROG.MAC heisst, dann lautet die Kommandozeile

```
ASM =PROG
```

und es wird eine uebersetzte Datei mit dem Namen

```
PROG.REL
```

erzeugt.

Als zusaetzliche Option kann nur ein Komma (,) auf der linken Seite des Gleichheitszeichens geschrieben werden. Dann werden vom Assembler alle Ausgabedateien (objekt und list) unterdrueckt.

Die Kommandozeile

```
ASM ,=PROG
```

veranlasst den Assembler, die Datei PROG.MAC zu uebersetzen, aber die erzeugten Dateien nicht auszugeben.
Die Programmierer verwenden diese Kommandozeile, um die Syntax im Quellprogramm zu testen, bevor das Uebersetzungsergebnis ausgegeben wird. Weil keine Dateien ausgegeben werden, ist der Assemblerlauf schneller beendet; die Fehler sind eher bekannt.

### 1.5.2.2. Objekt (dateiname)

Die Eintragung objekt ist immer optional.
Bestimmte Bedingungen zwingen dazu, objekt anzugeben.

Die Objektdatei schreibt das uebersetzte Programm in eine Diskettendatei. Diese wird vom Programmverbinder zum Erzeugen eines ausfuehrbaren Programms verwendet.

Wenn beide Eintragungen, objekt und list, in der Kommandozeile weggelassen sind (wie bei =source), erzeugt der Assembler eine Objektdatei mit dem gleichen Dateinamen wie die Quelldatei, aber mit der Standarderweiterung .REL.

Wenn ein vom Quelldateinamen verschiedener Name gewuenscht wird, muss dieser dateiname im Objekt-Feld eingetragen werden. Der Assembler haengt dann den Dateityp .REL an, wenn kein Dateityp eingetragen war.
Wenn beide Dateien - Objektkode und Druckdatei - gewuenscht werden, muessen beide Felder - objekt und list - ausgefuellt werden. Es wird nur eine Druckdatei erzeugt, wenn das Objekt-Feld frei bleibt.
Diese Form verwendet der Programmierer zum Testen der Syntax. Der Programmausdruck ist ausserdem eine Testhilfe.

Der Name fuer die Objektkodedatei kann jeder gueltige Dateiname sein. Es ist moeglich, einen anderen als den Quelldateinamen zu waehlen, aber es hat sich als guenstig erwiesen, alle Dateien, die zu einem Programm gehoeren, mit dem gleichen Namen zu versehen.

### 1.5.2.3. List (,dateiname)

Die Eintragung list ist immer optional. Das Komma vor der list-Eintragung muss geschrieben werden.
Wird eine Druckdatei gewuenscht, muss ein Dateiname in das list-Feld geschrieben werden (Eine Alternative zu dieser Regel siehe Pkt. 1.5.2.4.).

Der Assembler haengt .PRN als Standarddateityp an, wenn keine andere in der list-Eintragung angegeben wurde.

Die Kommandozeile

```
ASM ,PROG=PROG
```

uebersetzt die Datei PROG.MAC (Quelldatei) und erzeugt die Druckdatei PROG.PRN. Eine Objektdatei wird nicht erzeugt.

Die Druckdatei kann denselben Namen wie die Quelldatei haben oder jeden anderen gueltigen Dateinamen.

Wenn auf der linken Seite des Gleichheitszeichens nur ein Komma und keine Dateinamen stehen, uebersetzt der Assembler die Quelldatei, aber er erzeugt keine Ausgabedateien.

Das Kommando ASM ,=PROG
erlaubt die Pruefung der Quelldatei auf Syntaxfehler, bevor das uebersetzte Programm auf die Diskette geschrieben wird. Waehrend der Assembler die Pruefung der Fehler immer durchfuehrt, werden mit diesem Kommando keine Ausgabedateien erzeugt und damit ist der Assemblerlauf um einiges schneller.

Wenn die Uebersetzung beendet ist, gibt der Assembler folgende Meldung aus:

**[xx][No] Fatal error(s) [,xx warnings]**

Diese Meldung zeigt die Anzahl der schwerwiegenden Fehler und Warnungen, die im Programm aufgetreten sind, an.
Diese Meldung wird am Ende des Uebersetzungslaufes auf den Bildschirm und auf die Druckdatei ausgegeben.
Wenn diese Meldung erscheint, ist der Assembler fertig.
Heisst die Meldung "No fatal error(s)", so ist die Uebersetzung ohne Fehler durchlaufen.

Beispiele:

| | |
|---|---|
| ASM =TEST | Uebersetzen der Quellkodedatei TEST.MAC und Erzeugen der Objektkodedatei TEST.REL |
| ASM ,=TEST | Uebersetzen der Quellkodedatei TEST.MAC, ohne dass Ausgabedateien erzeugt werden. |
| ASM TEST,TEST=TEST | Uebersetzen der Quellkodedatei TEST.MAC, Erzeugen der Objektkode-Datei TEST.REL und der Druckdatei TEST.PRN |
| ASM OBJECT=TEST | Uebersetzen der Quellkodedatei und Erzeugen der Objektkodedatei OBJECT.REL. |

### 1.5.2.4. Schalter (/switch)

Das Assembler-Kommando kann neben dem Uebersetzen und Erzeugen von Objekt- und Druckdatei einige zusaetzliche Funktionen ausfuehren.

Diese zusaetzlichen Kommandos werden an das Ende der Kommandozeile eingegeben.
Eine solche Eingabe veranlasst den Assembler eine zusaetzliche oder alternative Funktion "einzuschalten": Diese Eingaben werden Schalter genannt.
Schalter sind Buchstaben, vor denen ein Schraegstrich (/) steht. Es kann jede beliebige Anzahl Schalter eingegeben werden, aber vor jedem Schalter muss ein Schraegstrich stehen.

Beispiel:

ASM ,=PROG/L/R

Folgende Schalter sind im Assembler moeglich:

| Schalter | Bedeutung |
|---|---|
| **/O** | Der Assembler druckt die Adressen in der Druckdatei oktal (zur Basis 8). |
| **/P** | Jedes /P legt einen extra Stack von 256 Bytes zur Verwendung waehrend der Uebersetzung an. /P sollte dann verwendet werden, wenn waehrend der Uebersetzung ein Stack-Ueberlauf-Fehler aufgetreten ist. Sonst ist er nicht notwendig. |
| **/R** | Erzwingt das Erzeugen einer Objektdatei mit dem gleichen Namen wie die Quelldatei. Das kann anstelle der Angabe des Dateinamens im Objekt-Feld der Kommandozeile verwendet werden. Die Angabe dieses Schalters ist geeignet, wenn eine .REL-datei erzeugt werden soll, aber vergessen wurde, einen Dateinamen in das Objekt-Feld einzutragen und ein Komma und ein Druckdateiname oder nur ein Komma vor dem Gleichheitszeichen geschrieben wurde. Also wenn geschrieben wurde:<br><br>`ASM ,PROG=PROG`<br>oder<br>`ASM ,=PROG`<br><br>dann kann auch eine .REL-Datei erzwungen werden durch einfaches Anfuegen eines /R vor dem Druecken von <ET>. Die Kommandozeile heisst dann:<br><br>`ASM ,PROG=PROG/R`<br>oder<br>`ASM ,=PROG/R` |
| **/X** | Der /X-Schalter setzt den Standard und die aktuelle Zuordnung auf Unterdruecken der Liste falscher Bedingungen. Nichtvorhandensein von /X bedeutet, der Standard und die aktuelle Zuordnung werden auf Drucken der falschen Bedingungen gesetzt. /X wird oft in Verbindung mit der bedingten Pseudooperation .TFCOND verwendet (Siehe Pkt. 1.4.1.5.). |
| **/L** | Erzwingt das Erzeugen einer Druckdatei mit dem gleichen Namen wie die Quelldatei. Wenn<br>`ASM =PROG`<br>oder<br>`ASM ,=PROG`<br>oder<br>`ASM PROG=PROG`<br>eingegeben wurde, dann kann durch einfaches Anhaengen eines /L vor dem Druecken von <ET> eine Druckdatei |

| Schalter | Bedeutung |
|---|---|

(PROG.PRN) erzwungen werden.
Die Kommandozeile heisst dann:

```
ASM =PROG/L
```

oder

```
ASM ,=PROG/L
```

oder

```
ASM PROG=PROG/L
```

**/C** Veranlasst den Assembler, eine spezielle Druckdatei (Cross-Referenz) mit dem gleichen Namen wie die Quelldatei anzulegen (fuer die Verwendung mit REF). (siehe auch Pkt. 1.7.)

**/Z** Veranlasst den Assembler, Z80-Operationskodes zu uebersetzen.
Dieser Schalter muss angegeben werden, wenn die Quelldatei im Z80-Kode geschrieben wurde, aber keine .Z80-Pseudooperation enthaelt.

**/I** Veranlasst den Assembler, 8080-Operationskodes zu uebersetzen.
Dieser Schalter muss angegeben werden, wenn die Quelldatei im 8080-Kode geschrieben wurde, aber keine .8080-Pseudooperation enthaelt.

**/H** Das ist der Standard.
Der Assembler druckt die Adressen in der Druckdatei hexadezimal.

**/M** Der /M-Schalter initialisiert einen Blockdatenbereich, der mit DS definiert wurde, auf ØØH.
Andererseits wird der Bereich nicht initialisiert. Das heisst, DS initialisiert den Speicherbereich nicht automatisch auf ØØH. In diesem Fall stehen beliebige Belegungen in dem reservierten Speicherbereich.

### 1.5.2.5. Zusaetzliche Angaben der Kommandozeile

In jedes Feld der Kommandozeile koennen noch zwei zusaetzliche Arten von Eintragungen vorgenommen werden - der Dateityp und die Geraetezuordnung. Diese beiden Arten sind aktuelle Teile einer "Dateispezifikation".
Eine Dateispezifikation besteht aus dem Geraet, wo die Datei lokalisiert ist, dem Namen der Datei und dem Dateityp.

Meist wird die Standardzuweisung fuer das Geraet und der Dateityp verwendet; sie wird dann vom Assembler eingefuegt, wenn diese Positionen in der Kommandozeile nicht ausgefuellt waren. Die Standardannahmen hindern nicht daran, entweder Dateityp

oder Geraetezuweisung einzugeben, einschliesslich der Eingaben, die den Standard darstellen.
Der Programmierer kann diese zusaetzlichen Eingaben in jeder Kombination angeben oder weglassen.

Das Format fuer eine Dateispezifikation ist:

**dev:dateiname.ext**

wobei

dev: 1-3 Buchstaben Geraetezuweisung (zwingend), gefolgt von einem Doppelpunkt

dateiname: 1-8 Zeichen Dateiname

.ext: 1-3 Zeichen Dateityp, dem (zwingend) ein Punkt vorausgehen muss.

**Dateityp (.ext)**

Zum Unterscheiden zwischen Quelldatei, Objektdatei und Druckdatei fuegt der Assembler einen Dateityp an den Dateinamen an. Der Dateityp ist eine aus 3 Zeichen bestehende Mnemonik, die an den Dateinamen angehaengt wird, wobei zwischen Dateinamen und Dateityp ein Punkt (.) steht.
Der vom Assembler ergaenzte Dateityp wird Standarddateityp genannt.

Die Standarddateitypen sind:

.REL Objektdatei
.PRN Druckdatei
.COM absolute (ausfuehrbare) Datei

Wenn fuer die Quelldatei kein Dateityp angegeben ist, nimmt der Assembler an, dass der Dateityp .MAC ist.

Es kann aber auch ein eigener Dateityp angegeben werden, wenn dies notwendig oder wuenschenswert ist. Es ist ein Nachteil, dass beim Rufen der Datei dann immer der Dateityp mit eingegeben werden muss. Bei Verwendung des Standarddateityps kann beim Ruf der Datei diese Angabe entfallen.

**Geraetezuweisung (dev:)**

Jedes Feld in der Kommandozeile kann auch eine Geraetezuweisung beinhalten.

Die im source-Feld spezifizierte Geraetezuweisung teilt dem Assembler mit, wo die Quelldatei zu finden ist. Die im Objekt-Feld oder list-Feld spezifizierte Geraetezuweisung sagt dem Assembler, wohin die Ausgabe der Objektdatei bzw. der Druckdatei erfolgen soll.
Wenn in einem der Felder die Geratezuweisung weggelassen ist, nimmt der Assembler standardmaessig das aktuell zugewiesene

Laufwerk an.
Das heisst, wenn das Geraet das aktuell zugewiesene Laufwerk ist, braucht die Geraetezuweisung nicht spezifiziert zu werden. Es ist notwendig, das Geraet zu spezifizieren, wenn ein anderes als das aktuell zugewiesene Laufwerk waehrend der Uebersetzung verwendet werden soll.

Beispiel:

Die Diskette mit dem Assembler liegt im Laufwerk A, die Programmdiskette im Laufwerk B. Die .REL-Datei soll ebenfalls auf B ausgegeben werden.

Dann muss in der Kommandozeile nur ASM =B:PROG
angegeben werden.

Wenn die .REL-Datei ausgegeben ist, ist das zugewiesene Laufwerk B. (Jedoch, wenn der Assembler beendet ist, ist A wieder das zugewiesene Laufwerk.)

Im Gegensatz dazu das folgende Beispiel:

Beispiel:

Die Quelldatei befindet sich mit auf der Diskette, auf der der Assembler ist. Diese Diskette liegt im Laufwerk A. Die .REL-Datei soll auf die im Laufwerk B liegende Diskette ausgegeben werden.

Die Kommandozeile lautet dann: ASM B:=A:PROG

Man sollte es sich zur Regel machen:

Wenn man nicht sicher ist, ob eine Geraetezuweisung angegeben werden muss, dann sollte man sie auf jeden Fall angeben. Das ist der sichere Weg, jede Datei auf den richtigen Platz zu bringen!

Gueltige Geraetezuweisungen fuer den Assembler sind:

| | |
|---|---|
| A:,B:,C:,... | Diskettenlaufwerke |
| LST: | Drucker |
| TTY: | Bildschirm oder Tastatur |

**Geraetezuweisung als Dateiname**

Eine weitere Option ist die Angabe der Geraetezuweisung ohne Dateiname in der Kommandozeile an die Stelle des Dateinames. Mit der Verwendung dieser Option werden verschiedene Resultate erzielt, abhaengig davon, welches Geraet in welches Feld eingetragen wird.

Beispiel: ASM ,TTY:=TTY:

ermoeglicht, eine unmittelbar eingegebene Zeile zu uebersetzen und auf Syntax- oder andere Fehler zu pruefen.

Aehnliches bewirkt: ASM ,LST:=TTY:

jedoch wird das Ergebnis nicht auf den Bildschirm, sondern auf den Drucker ausgegeben.

Wenn eines der Kommandos (,TTY:=TTY: oder ,LIST:=TTY:) verwendet wird, sollte als erstes eine END-Anweisung eingegeben werden. Der Assembler muss in den Pass 2 versetzt werden, bevor er Kode ausgibt.
Wenn einfach begonnen wird, Anweisungszeilen einzugeben ohne vorheriges END, wird keine Antwort kommen, bis eine END-Anweisung auftritt. Danach muessen alle Anweisungen erneut eingegeben werden, ehe der generierte Kode sichtbar wird.

Soll dieser Modus verlassen werden, so ist ^C einzugeben.

Tabelle 1 zeigt einige Moeglichkeiten, jedoch gibt sie keine vollstaendige Liste aller Kombinationen.

Tabelle 1: Wirkungen der Geraetezuweisung ohne Dateiname

| dev: | objekt | ,list | =source |
|---|---|---|---|
| A:,B:, C:,D:, | Schreiben der Datei auf das spezifizierte Laufwerk | Schreiben der Datei auf das spezifizierte Laufwerk | nicht moeglich (ein Dateiname muss spezifiziert werden) |
| LST: | nicht moeglich (unlesbares Dateiformat) | Schreiben der Liste auf den Drucker | nicht moeglich (nur Ausgabe) |
| TTY: | nicht moeglich (unlesbares Dateiformat) | "Schreiben" der Liste auf den Bildschirm | "Lesen" des Quellprogrammes von der Tastatur |

..
..

### 1.5.3. Format des Assemblerprotokolls

Die Druckliste des Assemblers besteht aus 2 Teilen in 2 verschiedenen Formaten, zum einen die Zeilen der Assemblerliste und zum anderen die Symboltabelle.

#### 1.5.3.1. Format der Assemblerliste

Jede Seite der Assemblerliste beginnt mit 2 Kopfzeilen. Wenn die Quelldatei keine Kopfzeilen aufweist (weder TITLE noch SUBTTL wurden angegeben), sind diese Teile der Kopfzeilen leer.

Das Format ist:

```
     [TITLE text]        ASM (SCPX-RBWS)  V n/n   SEITE  X
     [SUBTTL text]
```

Dabei sind:

TITLE text — der Text, der mittels der Pseudooperation TITLE in der Quelldatei angegeben wurde.
War in der Quelldatei keine TITLE-Pseudooperation vorhanden, dann ist dieser Platz in der Kopfzeile leer.

n/n — ist die Versionsnummer des Assemblers

X — ist die Seitennummer.
Diese wird nur angezeigt und erhoeht, wenn in der Quelldatei die Pseudooperation PAGE auftritt, oder wenn die gerade bearbeitete Seite gefuellt ist.

SUBTTL text — der Text, der mittels der Pseudooperation SUBTTL in der Quelldatei angegeben wurde.
War in der Quelldatei keine SUBTTL-Pseudooperation vorhanden, dann ist dieser Platz in der Kopfzeile leer.

Den beiden Kopfzeilen folgt eine Leerzeile.

In der darauf folgenden Zeile beginnt der Text der Druckdatei.

Das Format einer Druckzeile ist:

**[error] ####m xx xxxxm [w] text**

Dabei sind:

error — stellt einen Fehlerkode aus einem Zeichen bestehend dar.
Ein Fehlerkode wird nur gedruckt, wenn in dieser Zeile ein Fehler aufgetreten ist. Sonst bleibt der

Platz frei.

#### stellt den Zuordnungszaehler dar.
Die Zahl ist eine 4-stellige Hexadezimalzahl oder eine 6-stellige Oktalzahl.
Die Basis des Zuordnungszaehlers wird durch die Verwendung des /O- oder /H-Schalters im switch-Feld der Kommandozeile des Assemblers bestimmt.
Der Standard ist hexadezimal.

m zeigt den Zuordnungszaehlermodus an.
Die moeglichen Symbole dafuer sind:

| | |
|---|---|
| ' | Koderelativ |
| " | Datenrelativ |
| ! | Commonrelativ |
| <leer> | Absolut |
| * | External |

xx stellen den uebersetzten Kode dar.
xxxx xx ist ein 1-Byte-Wert. 1-Byte-Werten folgt immer ein Leerzeichen.
xxxx ist ein 2-Byte-Wert mit dem hoeherwertigen Byte zuerst! (Der Druck erfolgt im Gegensatz zur Reihenfolge der Abspeicherung von 2-Byte-Werten!)
2-Byte-Werten folgt ein Zeichen, das den Modus anzeigt (wie oben erlaeutert).
Dieses Zeichen wird durch das zweite m dargestellt.

[w] zeigt an, dass diese Zeile aus einer anderen Datei stammt.
Sie kann durch eine INCLUDE-Pseudooperation erzeugt worden sein oder Teil einer Makroerweiterung sein.
Eine Zeile aus einer INCLUDE-Anweisung erhaelt ein C, eine Zeile aus einer Makroerweiterung ein Pluszeichen (+). Sonst bleibt der Platz frei.

text steht fuer den Rest der Zeile, den Quelltext einschliesslich Marken, Operationskodes, Argumente und Kommentare.

### 1.5.3.2. Format der Symboltabelle

Die Seiten der Symboltabelle haben die gleichen Kopfzeilen wie die Assemblerliste. Aber anstelle der Seitennummer erscheint S.

Danach werden alle Makronamen, die im Programm verwendet werden, in alphabetischer Reihenfolge aufgelistet.
Im Anschluss daran erscheinen alle Symbole, ebenfalls in alphabetischer Reihenfolge.

Hinter jedem Symbol steht sein Wert, gefolgt von einem der nachfolgenden Zeichen:

I Public-Symbol
U Undefiniertes Symbol
C Commonblock-Name.
Der Wert, der hierfuer angzeigt wird, ist die Laenge des Blockes in Byte auf hexadezimaler oder oktalen Basis.
* EXTERNAL
' Koderelativer Wert
" Datenrelativer Wert
! Commonrelativer Wert

## 1.5.4. Fehlerkodes und Fehlermeldungen

Waehrend der Uebersetzung auftretende Fehler veranlassen den Assembler, entweder einen Fehlerkode oder eine Meldung zurueckzugeben. Fehlerkodes werden durch ein Zeichen dargestellt und in der Spalte 1 der Druckdatei gedruckt.

Auch wenn die Druckdatei nicht auf den Bildschirm ausgegeben wird, erscheinen die fehlerhaften Zeilen trotzdem auf dem Bildschirm.
Fehlermeldungen werden an das Ende der Druckdatei gedruckt oder am Ende nach den Fehlerzeilen auf dem Bildschirm angezeigt.

### 1.5.4.1. Fehlerkodes

| Fehler-kode | Bedeutung |
|---|---|
| **A** | **Argumentfehler**<br>Das zu der Pseudooperation gehoerige Argument befindet sich nicht im korrekten Format oder es liegt ausserhalb des zugelassenen Bereiches. |
| **C** | **Fehler der Bedingungsschachtelung**<br>ELSE ohne IF, ENDIF ohne IF, zweimal ELSE fuer ein IF, ENDC ohne COND. |
| **E** | **Externer Fehler**<br>Verwendung eines externen Symbols, das in diesem Zusammenhang nicht erlaubt ist.<br>(Bsp.: MARKE1 SET NAME##) |
| **M** | **Mehrfach definiertes Symbol**<br>Definition eines Symbols, das bereits definiert wurde. |

| Fehler-kode | Bedeutung |
|---|---|
| **N** | **Fehler einer Zahl**<br>Fehler in einer Zahl; meist ein falsches Zeichen, (z.B.: 8Q). |
| **O** | **Falscher Operationskode oder nicht einwandfreie Syntax**<br>SET, EQU oder MACRO ohne einen Namen; falsche Syntax in einem Operationskode oder falsche Syntax in einem Ausdruck (nichtpaarige runde Klammern, Anfuehrungszeichen, aufeinanderfolgende Operatoren usw.) |
| **P** | **Phasenfehler**<br>Der Wert einer Marke oder eines EQU-Namens ist im Pass 2 anders als im Pass 1. |
| **Q** | **Fragwuerdig**<br>Meistens ist eine Zeile nicht ordentlich abgeschlossen (bsp.: MOV AX,BX).<br>Dies ist eine Warnung. |
| **R** | **Verschiebung**<br>Nichterlaubte Verwendung einer Verschiebung in einem Ausdruck, (z.B.: abs-rel).<br>Daten, Kode und Common-Bereiche sind verschieblich. |
| **U** | **Undefiniertes Symbol**<br>Ein Symbol, auf das in einem anderen Ausdruck Bezug genommen wird, ist nicht definiert.<br>Fuer einige Pseudooperationen wird im Pass 1 ein V-Fehler erzeugt und im Pass 2 ein U-Fehler<br>(Siehe auch V-Fehler). |
| **V** | **Wertfehler**<br>Eine Pseudooperation (z.B.: .RADIX, .PAGE, DS, IF, IFE), hat im Pass 1 einen undefinierten Wert, obwohl er im Pass 1 bekannt sein muesste.<br>Wenn das Symbol spaeter im Program definiert wird, erzeugt der Assembler im Pass 2 keinen U-Fehler. |

### 1.5.4.2. Meldungen

**%No END statement**

Keine END-Anweisung.
Entweder sie fehlt oder sie wurde nicht durchlaufen infolge einer falschen Bedingung, eines nichtabgeschlossenen IRP-, IRPC-, REPT-Blockes oder eines abgeschlossenen Makros.

**Unterminated conditional**

Wenigstens eine Bedingung ist am Ende der Datei nicht abgeschlossen worden.

**Unterminated REPT/IRP/IRPC/MACRO**

Es ist wenigstens ein Block nicht abgeschlossen worden.

**Symbol table full**

Bei der Bildung der Symboltabelle durch den Assembler ist der verfuegbare Speicher erschoepft. Die haeufigste Ursache ist eine ganze Anzahl Makrobloecke, die Anweisungen fuer viele Anweisungszeilen enthalten.

Die Makrobloecke werden vollstaendig in der Symboltabelle abgespeichert, einschliesslich der Kommentare, die den Zeilen angehaengt sind innerhalb eines Makroblockes.

Es sollten also alle Makrobloecke im Quellprogramm geprueft werden.
Um die Kommentare innerhalb der Makrobloecke aus der Symboltabelle auszuschliessen, sollten vor die Kommentare zwei Semikolons (;;) geschrieben werden.

Dadurch sollte ausreichend Platz fuer die Uebersetzung des Programms freigeworden sein.

**{xx,No} Fatal error(s) [,xx warnings]**

Anzahl der schweren Fehler und Warnungen, die im Programm aufgetreten sind.
Diese Meldung wird am Ende jeder Uebersetzung auf den Bildschirm und in die Druckdatei ausgegeben.
Wenn diese Meldung erscheint, ist der Assemblerlauf beendet.

Die Meldung "**No Fatal error(s)**" zeigt an, dass die Uebersetzung vollstaendig und erfolgreich ist.

## 1.6. Beschreibung der Befehle

In diesem Kapitel wird die Wirkung der einzelnen CPU-Befehle beschrieben.
Der Abschnitt 1.6.16. enthaehlt das Abkuerzungsverzeichnis. Im Abschnitt 1.6.17. ist die Arbeit mit den Bedingungsbits (Flags) beschrieben.
Es gilt in der Beschreibung der Wirkungsweise der Befehle:

(HL) : Speicherplatz, der durch das Registerpaar HL adressiert wird. Register L beinhaltet dabei die niederwertigen 8 Bit und Register H die hoeherwertigen 8 Bit der Adresse.
(IX+d): Speicherplatz, der durch das Indexregister IX plus Verschiebung d adressiert wird.
(IY+d): Speicherplatz, der durch das Indexregister IY plus Verschiebung d adressiert wird.

### 1.6.1. Ladebefehle

Die Ladebefehle transportieren Daten intern zwischen den CPU-Registern oder zwischen CPU-Registern und dem Schreib- / Lesespeicher (RAM). Die Befehle muessen eine Ausgangsadresse, von der die Daten zu entnehmen sind, und eine Zieladresse enthalten.
Der Quellspeicherplatz wird durch den Ladebefehl nicht veraendert.

#### 1.6.1.1. 8-Bit-Ladebefehle

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| MOV r,r' | LD r,r' | Inhalt des Registers r' wird in das Register r umgespeichert. |
| MVI r,n | LD r,n | Der Direktoperand n wird in das Register r geladen. |
| MOV r,M | LD r,(HL) | Inhalt des Speicherplatzes M bzw. (HL) wird in das Register r geladen. |
| --- | LD r,(IX+d) | Inhalt des Speicherplatzes (IX+d) wird in das Register r geladen. |
| --- | LD r,(IY+d) | Inhalt des Speicherplatzes (IY+d) wird in das Register r geladen. |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| MOV M,r | LD (HL),r | Bringt ein Byte aus dem Register r auf den Speicherplatz M bzw. (HL). |
| --- | LD (IX+d),r | Bringt ein Byte aus dem Register r an den Speicherplatz (IX+d). |
| --- | LD (IY+d),r | Bringt ein Byte aus dem Register r an den Speicherplatz (IY+d). |
| MVI M,n | LD (HL),n | Bewirkt den Transport des mit n definierten Direktoperanden an den Speicherplatz M bzw. (HL). |
| --- | LD (IX+d),n | Bewirkt den Transport des mit n definierten Direktoperanden an den Speicherplatz (IX+d). |
| --- | LD (IY+d),n | Bewirkt den Transport des mit n definierten Direktoperanden an den Speicherplatz (IY+d). |
| LDAX B | LD A,(BC) | Der Inhalt des durch das Registerpaar BC adressierten Speicherplatzes wird in den Akkumulator geladen. Das C-Register beinhaltet die niederwertigen 8 Bit und das Register B die hoeherwertigen 8 Bit der Adresse. |
| LDAX D | LD A,(DE) | Der Inhalt des durch das Registerpaar DE adressierten Speicherplatzes wird in den Akkumulator geladen. Das Register E beinhaltet die niederwertigen 8 Bit und das Register D die hoeherwertigen 8 Bit der Adresse. |
| LDA nn | LD A,(nn) | Der Inhalt des durch nn adressierten Speicherplatzes wird in den Akkumulator geladen. |
| STA nn | LD (nn),A | Der Inhalt des Akkumulators wird auf den durch nn adressierten Speicherplatz geladen. |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| STAX B | LD (BC),A | Der Inhalt des Akkumulators wird auf den Speicherplatz geladen, dessen Adresse im Registerpaar BC definiert ist. Das Register C beinhaltet die niederwertigen 8 Bit und das Register B die hoeherwertigen 8 Bit der Adresse. |
| STAX D | LD (DE),A | Der Inhalt des Akkumulators wird auf den Speicherplatz geladen, dessen Adresse im Registerpaar DE definiert ist. Das Register E beinhaltet die niederwertigen 8 Bit und das Register D die hoeherwertigen 8 Bit der Adresse. |
| --- | LD A,I | Der Registerinhalt vom Interrupt-Register I wird in den Akkumulator geladen. |
| --- | LD A,R | Der Registerinhalt vom Refresh-Register R wird in den Akkumulator geladen. |
| --- | LD I,A | Der Inhalt des Akkumulators wird in das Interrupt-Register I geladen. |
| --- | LD R,A | Der Inhalt des Akkumulators wird in das Refresh-Register R geladen |

**1.6.1.2. 16-Bit-Ladebefehle**

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| LXI dd,nn | LD dd,nn | Ein 16-Bit-Direktoperand wird in ein Doppelregister dd geladen. |
| --- | LD IX,nn | Ein 16-Bit-Direktoperand wird in das Indexregister IX geladen. |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| --- | LD IY,nn | Ein 16-Bit-Direktoperand wird in das Indexregister IY geladen. |
| LHLD nn | LD HL,(nn) | Der Inhalt der durch nn und nn+1 adressierten Speicherplaetze wird in das Doppelregister HL geladen:<br>Inhalt von nn+1 ---> Register H<br>Inhalt von nn ---> Register L |
| --- | LD dd,(nn) | Der Inhalt der durch nn und nn+1 adressierten Speicherplaetze wird in ein Doppelregister dd geladen:<br>Inhalt von nn+1 ---> hoeherwertiges Register ($B,D,SP_H$)<br>Inhalt von nn ---> niederwertiges Register ($C,E,SP_L$) |
| --- | LD IX,(nn) | Der Inhalt der durch nn und nn+1 adressierten Speicherplaetze wird in das Indexregister IX geladen:<br>Inhalt von nn+1 ---> Register $IX_H$<br>Inhalt von nn ---> Register $IX_L$ |
| --- | LD IY,(nn) | Der Inhalt der durch nn und nn+1 adressierten Speicherplaetze wird in das Indexregister IY geladen:<br>Inhalt von nn+1 ---> Register $IY_H$<br>Inhalt von nn ---> Register $IY_L$ |
| SHLD nn | LD (nn),HL | Der Inhalt des Doppelregisters HL wird auf die Adressen nn und nn+1 transportiert:<br>Inhalt Reg. H --->Inhalt Adr.nn+1<br>Inhalt Reg. L --->Inhalt Adr.nn |
| --- | LD (nn),dd | Der Inhalt eines Registerpaares wird auf die Adressen nn und nn+1 transportiert:<br>Inhalt des hoeherwertigen Registers ($B,D,SP_H$) ---> Adr. nn+1<br>Inhalt des niederwertigen Registers ($C,E,SP_L$) ---> Adr. nn |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| --- | LD (nn),IX | Der Inhalt des Indexregisters IX wird auf die Adressen nn+1 und nn transportiert:<br>Inhalt $IX_H$ ---> Adr. nn+1<br>Inhalt $IX_L$ ---> Adr. nn |
| --- | LD (nn),IY | Der Inhalt des Indexregisters IY wird auf die Adressen nn+1 und nn transportiert:<br>Inhalt $IY_H$ ---> Adr. nn+1<br>Inhalt $IY_L$ ---> Adr. nn |
| SPHL | LD SP,HL | Der Inhalt des Doppelregisters HL wird in den Stackpointer (Kellerzeiger) uebertragen:<br>Inhalt Register H ---> $SP_H$<br>Inhalt Register L ---> $SP_L$ |
| --- | LD SP,IX | Der Inhalt des Indexregisters IX wird in den Stackpointer (Kellerzeiger) uebertragen.<br>Inhalt Register $IX_H$ ---> $SP_H$<br>Inhalt Register $IX_L$ ---> $SP_L$ |
| --- | LD SP,IY | Der Inhalt des Indexregisters IY wird in den Stackpointer (Kellerzeiger) uebertragen:<br>Inhalt Register $IY_H$ ---> $SP_H$<br>Inhalt Register $IY_L$ ---> $SP_L$ |

Anmerkung:
Das gegebenenfalls auf den Operationskode unmittelbar folgende Byte ist das niederwertige Byte des 16-Bit-Wortes.

## 1.6.2. Indirekte Registeroperationen (PUSH- und POP-Befehle)

### 1.6.2.1. PUSH-Befehle

Bei den PUSH-Befehlen wird der Inhalt des Registerpaares qq oder des Registers IX bzw. IY in einen externen RAM-Keller

(Stack) uebertragen, der als LIFO-Datei (letzte Eintragung wird zuerst gelesen) organisiert ist.
Der Stackpointer enthaelt dabei staendig eine aktuelle 16-Bit-Adresse, die der aktuell niedrigsten Adresse des Stackbereiches entspricht.

Der PUSH-Befehl subtrahiert 1 vom Inhalt des Stackpointers und laedt das hoeherwertige Byte des Registerpaares bzw. des Registers IX oder IY in die Speicherstelle, die durch den Inhalt des Stackpointers SP adressiert ist.
Danach wird der Inhalt des Stackpointers nochmals dekrementiert. Das niederwertige Byte wird jetzt in die Speicherstelle eingetragen, die durch den Inhalt des Stackpointers adressiert ist.

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| PUSH PSW | PUSH AF | Erniedrigen SP<br>(SP-1) <--- A<br>Erniedrigen SP<br>(SP-2) <--- F |
| PUSH B | PUSH BC | analog PUSH AF |
| PUSH D | PUSH DE | analog PUSH AF |
| PUSH H | PUSH HL | analog PUSH AF |
| --- | PUSH IX | (SP-2) <--- $IX_L$<br>(SP-1) <--- $IX_H$ |
| --- | PUSH IY | analog PUSH IX |

### 1.6.2.2. POP-Befehle

Bei den POP-Befehlen wird der Inhalt der vom Stackpointer (Kellerzeiger) SP und (SP+1) adressierten 2 Bytes des externen Stacks in ein Registerpaar qq bzw. in ein Register IX oder IY uebertragen.

Der POP-Befehl uebertraegt zunaechst den Inhalt der Speicherstelle, die durch den aktuellen Wert des Stackpointers adressiert ist, in den niederwertigen Teil des Registerpaares bzw. des Registers IX oder IY.
Danach wird der Stackpointer inkrementiert, und der Inhalt der jetzt adressierten Speicherstelle wird in den hoeherwertigen Teil des Registerpaares bzw. Registers IX oder IY uebertragen.
Der Stackpointer wird anschliessend erneut inkrementiert.

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| POP PSW | POP AF | F <--- (SP)<br>Erhoehen SP<br>A <--- (SP+1)<br>Erhoehen SP |
| POP B | POP BC | analog POP AF |
| POP D | POP DE | analog POP AF |
| POP H | POP HL | analog POP AF |
| --- | POP IX | $IX_H$ <--- (SP+1)<br>$IX_L$ <--- (SP) |
| --- | POP IY | analog POP IX |

Beispiel: Retten Register im Unterprogramm

```
BEISP: PUSH HL
       PUSH DE
       PUSH BC
         .        ) Unterprogramm-
         .        ) verarbeitungs-
         .        ) befehle
       POP  BC
       POP  DE
       POP  HL
       RET
```

Nach Verarbeitung des Unterprogramms besitzen die Register (HL, DE, BC) wieder die gleichen Inhalte wie vor dem Aufruf.

### 1.6.3. Register-Austausch-Befehle

Die Register-Austausch-Befehle sind ein Byte lang, ausgenommen der zwischen (SP) und jeweils einem Basisregister. Sie besitzen eine Befehlslaenge von 2 Byte.
Durch die geringe Befehlslaenge werden kurze Interruptantwortzeiten moeglich. Fuer den Registeraustausch steht ein Hintergrundsatz der CPU-Register zur Verfuegung (auch 2. Registersatz genannt).

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| XCHG | EX DE,HL | Die 16-Bit-Inhalte der Registerpaare DE und HL werden ausgetauscht:<br>DE <---> HL |
| --- | EX AF,AF' | Die 16-Bit-Inhalte der Registerpaare AF und AF' werden ausgetauscht. AF' besteht aus den Registern A' und F'.<br>AF <---> AF' |
| --- | EXX | Die 16-Bit-Inhalte der nachstehenden Register werden in folgender Weise getauscht:<br>BC <---> BC'<br>DE <---> DE'<br>HL <---> HL' |
| XTHL | EX (SP),HL | Der Inhalt des Registers L wird gegen den Inhalt der Speicherstelle ausgetauscht, die durch den Inhalt des Stackpointers SP adressiert ist. Der Inhalt des Registers H wird gegen den Inhalt der Speicherstelle ausgetauscht, die durch den Inhalt des Stackpointers SP plus 1 adressiert ist.<br>H <---> (SP+1)<br>L <---> (SP) |
| --- | EX (SP),IX | Der niederwertige Teil des Registers IX wird gegen den Inhalt der Speicherstelle ausgetauscht, die durch den Inhalt des Stackpointers SP adressiert ist. Der hoeherwertige Teil von IX wird gegen den Inhalt der Speicherstelle ausgetauscht, die durch den Inhalt SP plus 1 adressiert ist.<br>$IX_H$ <---> (SP+1)<br>$IX_L$ <---> (SP) |
| --- | EX (SP),IY | analog EX (SP),IX<br>$IY_H$ <---> (SP+1)<br>$IY_L$ <---> (SP) |

## 1.6.4. Blocktransportbefehle

Mit einem einzigen Befehl kann ein beliebig grosser Block des Speichers zu einem anderen Speicherplatz transportiert werden.

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| --- | LDIR | Transport mehrerer Datenbytes ab der Speicherstelle, die durch das Registerpaar HL adressiert wird, nach der Speicherstelle, die durch das Registerpaar DE adressiert wird. Die Bytezahl ist im Registerpaar BC enthalten. Nach jeder Byteuebertragung wird der Inhalt von HL und DE um 1 erhoeht und BC um 1 vermindert. Die Uebertragung endet, wenn (BC)=Ø ist.<br><br>Beispiel:<br>LD HL,DATA;Beginn Quellbereich<br>LD DE,PUF ;Beginn Zielbereich<br>LD BC,737 ;Laenge der Datenkette<br>LDIR ;Transport der Daten-<br>;kette nach Zielbereich |
| --- | LDI | Transport eines Datenbytes von der Speicherstelle, die durch das Register HL adressiert wird, nach der Speicherstelle, die durch das Register DE adressiert wird. Die Register DE und HL werden um 1 erhoeht, und das Register BC wird um 1 vermindert.<br><br>Beispiel:<br>LD HL,DATA;Beginn Quellbereich<br>LD DE,PUF ;Beginn Zielbereich<br>LD BC,132 ;Max. Kettenlaenge BC<br>LD A,'K' ;Endemerkmal im Reg. A<br>LOOP:CMP (HL);Vergleich Speicher-<br>;inhalt mit Endemerkm.<br>JR Z,END-$;Sprung zu END, wenn<br>;Zeichen gleich<br>LDI ;Zeichentransport (HL)<br>;nach (DE)<br>;Erhoehen HL und DE,<br>;Vermindern BC<br>JP PE,LOOP;Sprung zu LOOP, wenn<br>;noch Zeichen zu trans- |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| | | ;portieren sind, sonst<br>;gehe weiter.<br>;P/V-Flag=1, solange<br>;BC≠Ø ist.<br>END: JP HALT; Halt |
| --- | LDDR | Der Befehl wirkt wie LDIR, nur werden hier die Register DE und HL um 1 vermindert. |
| --- | LDD | Der Befehl wirkt wie LDI, nur werden hier die Register DE und HL um 1 vermindert. |

### 1.6.5. Blocksuchbefehle

Diese Befehle sind fuer die Verarbeitung grosser Datenmengen geeignet. Mit einem einzigen Befehl kann ein Speicherblock beliebiger Groesse nach einem bestimmten 8-Bit-Zeichen durchsucht werden. Der Befehl ist automatisch beendet, wenn das Zeichen gefunden wurde.

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| --- | CPI | Vergleich des Inhalts des durch HL adressierten Speicherplatzes mit dem Inhalt des Akkumulators (A-Register). Das Register BC kann als Bytezaehler arbeiten. Das Register HL wird um 1 erhoeht. Das Registerpaar BC wird um 1 vermindert. |
| --- | CPIR | Vergleich des Inhalts des Akkumulators mit dem Inhalt eines adressierten Speicherbereiches. Die Startadresse des Bereiches ist in dem Registerpaar HL enthalten, die Laenge des Bereiches in dem Registerpaar BC. Die zu suchende Konstante steht im Akkumulator. Der Vergleich endet, wenn der |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| | | Akkumulator = (HL) ist oder wenn BC = Ø ist. Der Befehl sucht, indem er Register HL erhoeht und Register BC um 1 vermindert. |
| --- | CPD | Der Befehl wirkt wie CPI, nur wird hier das Register HL vermindert. |
| --- | CPDR | Der Befehl wirkt wie CPIR, nur wird hier das Register HL vermindert. |

## 1.6.6. Arithmetische und logische Operationen

Die arithmetischen und logischen Befehle arbeiten mit Daten, die sich im Akkumulator (A-Register) und in anderen Universal-CPU-Registern oder auf den Speicherplaetzen befinden.
Die Ergebnisse nach Ausfuehrung dieser Operationen stehen im Akkumulator. Die Flags werden entsprechend dem Ergebnis der Operationen gesetzt.

### 1.6.6.1. 8-Bit-Arithmetik

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| ADD r | ADD A,r | Der Registerinhalt r wird zum Akkumulatorinhalt addiert. |
| ADD M | ADD A,(HL) | Der Inhalt des Speicherplatzes M bzw. (HL) wird zum Inhalt des Akkumulators addiert. |
| ADI n | ADD A,n | Der Direktoperand n wird zum Inhalt des Akkumulators addiert. |
| --- | ADD A,(IX+d) | Der Inhalt des Speicherplatzes (IX+d) wird zum Inhalt des Akkumulators addiert. |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| --- | ADD A,(IY+d) | Der Inhalt des Speicherplatzes (IY+d) wird zum Inhalt des Akkumulators addiert. |
| ADC r | ADC A,r | Der Registerinhalt r und Carry-Flag (CY) werden zum Akkumulatorinhalt addiert. |
| ADC M | ADC A,(HL) | Der Inhalt des Speicherplatzes M bzw. (HL) und Carry-Flag (CY) werden zum Inhalt des Akkumulators addiert. |
| ACI n | ADC A,n | Der Direktoperand n und CY werden zum Inhalt des Akkumulators addiert. |
| --- | ADC A,(IX+d) | Der Inhalt des Speicherplatzes (IX+d) und CY werden zum Inhalt des Akkumulators addiert. |
| --- | ADC A,(IY+d) | Der Inhalt des Speicherplatzes (IY+d) und CY werden zum Inhalt des Akkumulators addiert. |
| SUB r | SUB r | Der Registerinhalt r wird vom Inhalt des Akkumulators subtrahiert |
| SUB M | SUB (HL) | Der Inhalt des Speicherplatzes M bzw. (HL) wird vom Inhalt des Akkumulators subtrahiert. |
| SUI n | SUB n | Der Direktoperand n wird vom Inhalt des Akkumulators subtrahiert |
| --- | SUB (IX+d) | Der Inhalt des Speicherplatzes (IX+d) wird vom Inhalt des Akkumulators subtrahiert. |
| --- | SUB (IY+d) | Der Inhalt des Speicherplatzes (IY+d) wird vom Inhalt des Akkumulators subtrahiert. |
| SBB r | SBC A,r | Der Registerinhalt r und Carry-Flag (CY) werden vom Inhalt des Akkumulators subtrahiert. |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| SBB M | SBC A,(HL) | Der Inhalt des Speicherplatzes M bzw. (HL) und Carry-Flag (CY) werden vom Inhalt des Akkumulators subtrahiert. |
| SBI n | SBC A,n | Der Direktoperand n und Carry-Flag (CY) werden vom Inhalt des Akkumulators subtrahiert. |
| --- | SBC A,(IX+d) | Der Inhalt des Speicherplatzes (IX+d) und Carry-Flag (CY) werden vom Inhalt des Akkumulators subtrahiert. |
| --- | SBC A,(IY+d) | Der Inhalt des Speicherplatzes (IY+d) und Carry-Flag (CY) werden vom Inhalt des Akkumulators subtrahiert. |
| INR r | INC r | Der Inhalt des Registers r wird um 1 erhoeht. |
| INR M | INC (HL) | Der Inhalt des Speicherplatzes M bzw. (HL) wird um 1 erhoeht. |
| --- | INC (IX+d) | Der Inhalt des Speicherplatzes (IX+d) wird um 1 erhoeht. |
| --- | INC (IY+d) | Der Inhalt des Speicherplatzes (IY+d) wird um 1 erhoeht. |
| DCR r | DEC r | Der Inhalt des Registers r wird um 1 vermindert. |
| DCR M | DEC (HL) | Der Inhalt des Speicherplatzes M bzw. (HL) wird um 1 vermindert. |
| --- | DEC (IX+d) | Der Inhalt des Speicherplatzes (IX+d) wird um 1 vermindert. |
| --- | DEC (IY+d) | Der Inhalt des Speicherplatzes (IY+d) wird um 1 vermindert. |

### 1.6.6.2. 16-Bit-Arithmetik

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| DAD dd | ADD HL,dd | Der Inhalt von dd wird zum Inhalt des Registerpaares HL addiert. |
| --- | ADD IX,IX | Der Inhalt des Registers IX wird mit sich selbst addiert. Diese Verdoppelung ist gleichbedeutend mit einer Linksverschiebung der 16 Bit um eine Bitposition. |
| --- | ADD IY,IY | Der Inhalt des Registers IY wird mit sich selbst addiert. Diese Verdoppelung ist gleichbedeutend mit einer Linksverschiebung der 16 Bit um eine Bitposition. |
| --- | ADD IX,pp | Der Inhalt von pp wird zum Inhalt des 16-Bit-Registers IX addiert. |
| --- | ADD IY,pp | Der Inhalt von pp wird zum Inhalt des 16-Bit-Registers IY addiert. |
| --- | ADC HL,dd | Der Inhalt des Doppelregisters dd und das Carry-Flags (CY) werden zum Inhalt des Registerpaares HL addiert. |
| --- | SBC HL,dd | Der Inhalt des Doppelregisters dd und das Carry-Flag (CY) werden vom Inhalt des Registerpaares HL subtrahiert. |
| INX dd | INC bb | Der Inhalt des Doppelregisters dd bzw. bb wird um 1 erhoeht. |
| DCX dd | DEC bb | Der Inhalt des Doppelregisters dd bzw. bb wird um 1 vermindert. |

## 1.6.6.3. 8-Bit-Logikbefehle

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| ANA r<br>ANA M<br>ANI n | AND s<br>s = r,n,(HL),<br>(IX+d),(IY+d) | Logisches UND eines Registers, Speicherbytes oder Direktwertes mit dem Akkumulator. Das spezifizierte Byte wird bitweise mit dem Inhalt des Akkumulators konjunktiv verknuepft. Das logische UND zweier Bits ist nur dann 1, wenn beide Bits 1 sind.<br>Beispiel:<br>Akkumulator: FCH 1111 11ØØ<br>Register: ØFH ØØØØ 1111<br>--------------<br>Resultat: ØCH ØØØØ 11ØØ |
| ORA r<br>ORA M<br>ORI n | OR s<br>s = r,n,(HL),<br>(IX+d),(IY+d) | Logisches ODER eines Registers, Speicherbytes oder Direktwertes mit dem Akkumulator. Das spezifizierte Byte wird bitweise mit dem Inhalt des Akkumulators disjunktiv verknuepft. Das logische ODER zweier Bits ist nur dann Ø, wenn beide Bits Ø sind.<br>Beispiel:<br>Akkumulator: FCH 1111 11ØØ<br>Register: F1H 1111 ØØØ1<br>--------------<br>Resultat: FDH 1111 11Ø1 |
| XRA r<br>XRA M<br>XRI n | XOR s<br>s = r,n,(HL),<br>(IX+d),(IY+d) | Exklusives ODER eines Registers, Speicherbytes oder Direktwertes mit dem Akkumulator. Das spezifizierte Byte wird bitweise mit dem Inhalt des Akkumulators exklusiv verknuepft. Das exklusive ODER ist dann gleich 1, wenn ein Bit=1 und ein Bit=Ø ist.<br>Beispiel:<br>Akkumulator: FCH 1111 11ØØ<br>Register: F1H 1111 ØØØ1<br>--------------<br>Resultat: ØDH ØØØØ 11Ø1 |
| CMP r<br>CMP M<br>CPI n | CP s<br>s = r,n,(HL),<br>(IX+d),(IY+d) | Der Inhalt von s wird mit dem Akkumulator verglichen. Der urspruengliche Inhalt von A bleibt erhalten. Das Vergleichsergebnis ist durch Flags erkennbar. |

## 1.6.7. Sprungbefehle

Es ist zwischen unbedingten und bedingten Spruengen zu unterscheiden. Es sind weiterhin relative Spruenge moeglich, die zur Adressenbildung anstelle von zwei Bytes nur eines benoetigen.

Bei bedingten Spruengen werden Sprungbedingungen getestet. Diese Bedingungen sind im Flagregister F enthalten.
In Abhaengigkeit von den Bedingungsflags koennen die Sprungbedingungen erfuellt sein oder nicht.
Bei einer erfuellten Sprungbedingung wird der Speicherzuordnungszaehler entsprechend der Adressenangabe im Sprungbefehl veraendert.
Bei nicht erfuellter Sprungbedingung wird der Sprungbefehl ignoriert.
Bei den relativen Spruengen wird das Sprungziel ueber den Wert e errechnet. Die Sprungweite e wird zum aktuellen Stand des Speicherzuordnungszaehlers addiert (im 2er-Komplement) und ermoeglicht einen Sprung im Bereich zwischen -128 und 127 Bytes.
Bei symbolischer Adressierung in relativen Spruengen berechnet der Assembler automatisch den Speicherzuordnungszaehler.

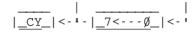
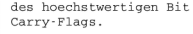
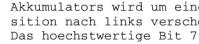
| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| JMP nn | JP nn | Unbedingter Sprung nach Adresse nn |
| JNZ nn | JP NZ,nn | Sprung nach Adresse nn, wenn Z-Flag gleich Ø ist. |
| JZ nn | JP Z,nn | Sprung nach Adresse nn, wenn Z-Flag gleich 1 ist. |
| JNC nn | JP NC,nn | Sprung nach Adresse nn, wenn C-Flag gleich Ø ist. |
| JC nn | JP C,nn | Sprung nach Adresse nn, wenn C-Flag gleich 1 ist. |
| JPO nn | JP PO,nn | Sprung nach Adresse nn, wenn P/V-Flag gleich Ø ist. |
| JPE nn | JP PE,nn | Sprung nach Adresse nn, wenn P/V-Flag gleich 1 ist. |
| JP nn | JP P,nn | Sprung nach Adresse nn, wenn S-Flag gleich Ø ist. |

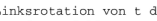
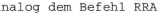
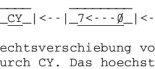
| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| JM nn | JP M,nn | Sprung nach Adresse nn, wenn S-Flag 1 ist. |
| --- | JR e | Unbedingter relativer Sprung. |
| --- | JR NZ,e | Relativer Sprung um Verschiebung e, wenn Z-Flag gleich Ø ist. |
| --- | JR Z,e | Relativer Sprung um Verschiebung e, wenn Z-Flag gleich 1 ist. |
| --- | JR NC,e | Relativer Sprung um Verschiebung e, wenn C-Flag gleich Ø ist. |
| --- | JR C,e | Relativer Sprung um Verschiebung e, wenn C-Flag gleich 1 ist. |
| PCHL | JP (HL) | Unbedingter Sprung zur Adresse, die im Register HL steht. |
| --- | JP (IX) | Unbedingter Sprung zur Adresse, die im Register IX steht. |
| --- | JP (IY) | Unbedingter Sprung zur Adresse, die im Register IY steht. |
| --- | DJNZ e | Der Inhalt des Registers B wird um 1 vermindert. Bedingter relativer Sprungbefehl um Verschiebung e, wenn der Inhalt des Registers B ≠ Ø ist. |

### 1.6.8. Verschiebebefehle

Durch diese Befehle ist die Moeglichkeit gegeben, im Akkumulator, in einem Universalregister oder in einem Speicherplatz Daten einfach oder zyklisch zu verschieben. Diese Operationen sind in einem externen grossen Gebiet einschliesslich der ganzzahligen Multiplikation und Division anwendbar.

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| RLC | RLCA | Linksrotation des Akkumulatorinhalts. Der Inhalt des Akkumula- |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| | | tors wird um eine Bitposition nach links verschoben. Das hoechstwertige Bit 7 wird zum Inhalt des niederwertigen Bits Ø und des Carry-Flags.  |
| RRC | RRCA | Rechtsrotation des Akkumulatorinhalts. Der Inhalt des Akkumulators wird um eine Bitposition nach rechts verschoben. Das niederwertige Bit Ø wird zum Inhalt des hoechstwertigen Bit 7 und des Carry-Flags.  |
| RAL | RLA | Linksrotation des Akkumulatorinhalts durch CY. Der Inhalt des Akkumulators wird um eine Bitposition nach links verschoben. Das hoechstwertige Bit 7 ersetzt das Carry-Flag, waehrend das Carry-Flag das niederwertigste Bit Ø des Akkumulators ersetzt.  |
| RAR | RRA | Rechtsrotation des Akkumulatorinhalts durch CY. Der Inhalt des Akkumulators wird um eine Bitposition nach rechts verschoben. Das niederwertigste Bit Ø ersetzt des Carry-Flag, waehrend das hoechstwertige Bit 7 durch das Carry-Flag ersetzt wird.  |
| --- | RLC t<br>t=r,(HL),<br>(IX+d),(IY+d) | Linksrotation von t analog dem Befehl RLCA. |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| --- | RRC t | Rechtsrotation von t analog dem Befehl RRCA. |
| --- | RL t | Linksrotation von t durch CY analog dem Befehl RLA. |
| --- | RR t | Rechtsrotation von t durch CY analog dem Befehl RRA. |
| --- | SLA t | Linksverschiebung von t um 1 Bit durch CY. Das niederwertige Registerbit Ø wird Ø.<br> |
| --- | SRL t | Rechtsverschiebung von t um 1 Bit durch CY. Das hoechstwertige Registerbit 7 wird Ø.<br>  |
| --- | SRA t | Rechtsverschiebung von t um 1 Bit durch CY. Der Inhalt von Bit 7 bleibt erhalten.<br>  |
| --- | RLD | Zyklische Verschiebung nach links zwischen dem Akkumulator und dem Inhalt des durch HL adressierten Speicherplatzes.<br>  <br>Die unteren 4 Bit des durch HL adressierten Speicherplatzes werden in die oberen 4 Bitstellen uebertragen und diese ihrerseits in die unteren 4 Bitstellen des Akkumulators. Die unteren 4 Bits des Akkumulators werden in die unteren 4 Bits der Speicherstelle transportiert. |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| --- | RRD | Zyklische Verschiebung nach rechts zwischen dem Akkumulator und dem Inhalt des durch HL adressierten Speicherplatzes.<br><br>**A** **(HL)**<br>\|Ø1Ø1\|1111\| \|ØØØØ\|1111\|<br>vor dem Befehl<br>\|Ø1Ø1\|1111\| \|1111\|ØØØØ\|<br>nach dem Befehl<br><br>Die unteren 4 Bits der durch HL adressierten Speicherstelle werden in die unteren 4 Bitstellen des Akkumulators uebertragen und diese in die oberen der durch HL adressierten Speicherstelle. Die oberen 4 Bits aus der durch HL adressierten Speicherstelle werden in die unteren 4 Bitstellen transportiert. |

## 1.6.9. Spezielle Akkumulator- und Flagbefehle

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| DAA | DAA | Korrigiert nach Addition / Subtraktion zweier gepackter BCD-Zahlen den Akkumulatorinhalt so, dass im Akkumulator wieder die gepackte BCD-Darstellung erreicht wird.<br>Beispiel:<br><br>65 Ø11Ø Ø1Ø1<br>+57 Ø1Ø1 Ø111<br>---<br>1Ø11 11ØØ<br>=== |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| | | Der DAA-Befehl fuehrt jetzt die notwendige Korrektur aus. In Abhaengigkeit von der Wertigkeit der zwei Halbbytes wird ein bestimmtes Korrekturbyte (im vorliegenden Beispiel 66) addiert.<br><br>CY \| Ø11Ø Ø11Ø<br>Ÿ \|----------<br>L \| ØØ1Ø ØØ1Ø = 122 |
| CMA | CPL | Bitweises Negieren (Komplementieren) des Akkumulatorinhaltes. |
| --- | NEG | Subtrahieren des Akkumulator-Inhalts von Ø. Entspricht wertmaessig dem Zweierkomplement. |
| CMC | CCF | Komplementieren des Carry-Flags. |
| STC | SCF | Setzen des Carry-Flags. |

## 1.6.10. Unterprogramm-Aufruf-Befehle

Diese Befehle sind eine Spezialform der Sprungbefehle. Es ist zwischen unbedingten und bedingten Unterprogrammaufrufen zu unterscheiden. Beim unbedingten Unterprogrammaufruf wird der dem Aufruf folgende Speicherplatzzuordnungszaehlerstand in den Stack gerettet.

Die im Befehl angegebene Unterprogrammstartadresse nn wird vom Speicherplatzzuordnungszaehler wieder mit der Absprungadresse aus dem Stack geladen.
Bei bedingten Unterprogrammaufrufen wird bei erfuellter Sprungbedingung analog dem unbedingten Unterprogrammaufruf verfahren. Bei nicht erfuellter Sprungbedingung wird der Befehl ignoriert.
Der hoeherwertige Adressteil im Speicherplatzzuordnungszaehler wird nach der Stackadresse minus 1 und der niederwertige Adressteil nach der Stackadresse minus 2 gebracht.

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| CALL nn | CALL nn | Unbedingter Unterprogrammaufruf<br>(SP-1) <--- $PC_H$<br>(SP-2) <--- $PC_L$<br>PC <--- nn |
| CNZ nn | CALL NZ,nn | Unterprogrammaufruf, wenn Z-Flag gleich Ø ist. |
| CZ nn | CALL Z,nn | Unterprogrammaufruf, wenn Z-Flag gleich 1 ist. |
| CNC nn | CALL NC,nn | Unterprogrammaufruf, wenn das C-Flag gleich Ø ist. |
| CC nn | CALL C,nn | Unterprogrammaufruf, wenn das C-Flag gleich 1 ist. |
| CPO nn | CALL PO,nn | Unterprogrammaufruf, wenn das P/V-Flag gleich Ø ist. |
| CPE nn | CALL PE,nn | Unterprogrammaufruf, wenn das P/V-Flag gleich 1 ist. |
| CP nn | CALL P,nn | Unterprogrammaufruf, wenn das S-Flag gleich Ø ist. |
| CM nn | CALL M,nn | Unterprogrammaufruf, wenn das S-Flag gleich 1 ist. |
| RST k<br>k=0,1,2,3,4,5,6,7 | RST p | Der RST-Befehl ist ein spezieller Unterprogrammaufruf. Es sind folgenden 8 RST-Adressen zugelassen: p={ØØH; Ø8H; 1ØH; 18H; 28H; 3ØH; 38H}.<br>Der hoeherwertige Adressteil ist dabei Ø.<br>Der RST-Befehl entspricht in der weiteren Wirkung dem unbedingten Unterprogrammaufruf. |

### 1.6.11. Unterprogramm-Ruecksprung-Befehle

Ein Ruecksprungbefehl beendet ein Unterprogramm. Es wird zwischen einem unbedingten Ruecksprung, bedingten Rueckspruengen und Rueckspruengen aus Interrupt-Behandlungsroutinen unterschieden.
Bei einem unbedingten Ruecksprung und bei erfuellter Sprungbedingung bei bedingten Rueckspruengen wird der beim Aufruf des Unterprogramms in den Stack gerettete Speicherzuordnungszaehlerinhalt wieder in den Speicherzuordnungszaehler zurueckgeschrieben.

$PC_L$ <--- (SP)

$PC_H$ <--- (SP+1)

SP <--- (SP+2)

Bei nichterfuellter Sprungbedingung wird der dem Ruecksprung folgende Befehl abgearbeitet.

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| RET | RET | Unbedingter Ruecksprung |
| RNZ | RET NZ | Unterprogrammruecksprung, wenn das Z-Flag gleich Ø ist. |
| RZ | RET Z | Unterprogrammruecksprung, wenn das Z-Flag gleich 1 ist. |
| RNC | RET NC | Unterprogrammruecksprung, wenn das C-Flag gleich Ø ist. |
| RC | RET C | Unterprogrammruecksprung, wenn das C-Flag gleich 1 ist. |
| RPO | RET PO | Unterprogrammruecksprung, wenn das P/V-Flag gleich Ø ist. |
| RPE | RET PE | Unterprogrammruecksprung, wenn das P/V-Flag gleich 1 ist. |
| RP | RET P | Unterprogrammruecksprung, wenn das S-Flag gleich Ø ist. |
| RM | RET M | Unterprogrammruecksprung, wenn das S-Flag gleich 1 ist. |
| --- | RETI | Es erfolgt ein Ruecksprung aus einer Interrupt-Behandlungsroutine. Dem Peripheriebaustein, der |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| | | das Interrupt anmeldete, wird das Ende seines Programms mitgeteilt. Der Baustein gibt daraufhin die von ihm blockierte Interrupt-Kette (DAISY-CHAIN) wieder frei und ermoeglicht damit die Abarbeitung niederwertiger Interrupts. Der Inhalt von IFF2 wird nach IFF1 uebertragen. Durch die RETI-Anweisung wird der maskierbare Interrupt nicht freigegeben. Es sollte grundsaetzlich vor jeder RETI-Anweisung ein EI-Befehl stehen, der die Annahme spaeter folgender Interruptanforderungen ermoeglicht. |
| --- | RETN | Es erfolgt ein Ruecksprung aus einer Interrupt-Behandlungsroutine, die durch einen nichtmaskierbaren Interrupt (NMI) ausgeloest wurde. Die Anweisung wirkt zunaechst wie die RET-Anweisung. Zusaetzlich wird der Inhalt von IFF2 nach IFF1 uebertragen, so dass die Abarbeitung maskierbarer Interrupt-Anforderungen unmittelbar nach Ausfuehrung des RETN-Befehls freigegeben ist, falls sie vor der NMI-Anforderung freigeben war. |

## 1.6.12. CPU-Steuerbefehle

Diese Steuerbefehle loesen bei der CPU verschiedene Bedingungen und Betriebsarten aus.
Diese Gruppe enthaelt auch solche Befehle wie das Ein- und Ausschalten des Interrupt-Annahme-Flip-Flops oder das Setzen der Betriebsart Interruptverhalten.

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| NOP | NOP | Die CPU fuehrt keine Operationen |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| | | aus. Es werden Refresh-Zyklen erzeugt. |
| HLT | HALT | Die CPU fuehrt solange eine Folge von NOP-Befehlen aus, bis ein Interrupt oder der RESET-Eingang an der CPU aktiv wird. Es werden Refresh-Zyklen erzeugt. |
| DI | DI | Der maskierbare Interrupt wird durch Ruecksetzen der Interrupt-Freigabe-Flip-Flops IFF1 bzw. IFF2 der CPU gesperrt. Nichtmaskierbare Interrupts werden anerkannt. |
| EI | EI | Der maskierbare Interrupt wird durch Setzen der Interrupt-Freigabe-Flip-Flops IFF1 bzw. IFF2 der CPU freigegeben. Waehrend der Ausfuehrung des Befehls akzeptiert die CPU keine Interruptanforderungen. |
| --- | IM Ø | Der Befehl bringt die CPU in den Interruptmodus Ø. |
| --- | IM 1 | Der Befehl bringt die CPU in den Interruptmodus 1. |
| --- | IM 2 | Der Befehl bringt die CPU in den Interruptmodus 2. |

### 1.6.13. Bittest- und -Setzbefehle (Bitmanipulation)

Die Bitmanipulationsbefehle erlauben, Bits in einem Register oder auf einem Speicherplatz zu setzen, zu loeschen und zu testen.

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| --- | SET b,r | Die durch b gekennzeichnete Bitposition wird in dem Register r gesetzt. |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| --- | SET b,(HL) | Die durch b gekennzeichnete Bitposition wird in der Speicherstelle (HL) gesetzt. |
| --- | SET b,(IX+d) | Die durch b gekennzeichnete Bitposition wird in der Speicherstelle (IX+d) gesetzt. |
| --- | SET b,(IY+d) | Die durch b gekennzeichnete Bitposition wird in der Speicherstelle (IY+d) gesetzt. |
| --- | RES b,t<br>t=r,(HL),<br>(IX+d),(IY+d) | Die durch b gekennzeichnete Bitposition in t wird geloescht. |
| --- | BIT b,t | Die durch b gekennzeichnete Bitposition in t wird getestet. Das Komplement des zu testenden Bits wird in das Z-Flag geladen. |

## 1.6.14. Eingabebefehle

Die Ein- und Ausgabegruppe gestattet einen weiteren Anwendungsbereich von Datentransfer zwischen Speicherplaetzen oder den Universalregistern der CPU und den externen E/A-Geraeten. Die Eingabebefehle setzen automatisch das Flagregister, so dass keine zusaetzlichen Befehle noetig sind, um den Status der Eingabedaten zu ermitteln.

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| IN n | IN A,(n) | Kanaladresse wird mittels Direktoperand eingestellt.<br>Zielregister ist der Akkumulator<br>A <--- (n) |
| --- | IN r,(C) | Kanaladresse wird indirekt ueber das Register C eingestellt.<br>r <--- (C) |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| --- | INI | Kanaladresse wird indirekt ueber Register C eingestellt. Zieladr. ueber Register HL. B kann als Bytezaehler arbeiten. B wird dekrementiert, HL inkrementiert.<br>(HL) <--- (C)<br>B <--- B-1<br>HL <--- HL+1 |
| --- | INIR | Kanaladresse wird indirekt ueber Register C eingestellt, Zieladr. ueber Register HL. B arbeitet als Bytezaehler. B wird dekrementiert, HL inkrementiert. Es wird eine Blockuebertragung durchgefuehrt bis B=Ø ist.<br>(HL) <--- (C)<br>B <--- B-1<br>HL <--- HL+1<br>Wiederholen bis B=Ø. |
| --- | IND | Kanaladresse wird indirekt ueber Register C eingestellt. Zieladr. ueber Register HL. B kann als Bytezaehler arbeiten. B und HL werden dekrementiert.<br>(HL) <--- (C)<br>B <--- B-1<br>HL <--- HL-1 |
| --- | INDR | Kanaladresse wird indirekt ueber Register C eingestellt. Zieladr. ueber Register HL. B arbeitet als Bytezaehler. B und HL werden dekrementiert. Es wird eine Blockuebertragung durchgefuehrt, bis B = Ø ist.<br>(HL) <--- (C)<br>B <--- B-1<br>HL <--- HL-1<br>Wiederholen B = Ø. |

Die Kanaladresse liegt auf der unteren Haelfte des Adressenbusses AØ - A7. Auf der oberen Haelfte des Adressenbusses (A8 - A15) liegt bei IN A,(n) der Akkumulatorinhalt, bei den restlichen Befehlen der Inhalt des Registers B.

## 1.6.15. Ausgabebefehle

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| OUT n | OUT (n),A | Kanaladresse wird mit Direktoperand eingestellt. Quellregister ist der Akkumulator: (n) <--- A. |
| --- | OUT C,r | Kanaladresse wird indirekt ueber Register C eingestellt. Quellregister ist r: (C) <--- r. |
| --- | OUTI | Kanaladresse wird indirekt ueber das Register C eingestellt, Quelladresse ueber Register HL. B kann als Bytezaehler arbeiten. B wird dekrementiert, HL inkrementiert.<br>(C) <--- (HL)<br>B <--- B-1<br>HL <--- HL+1 |
| --- | OTIR | Kanaladresse wird indirekt ueber das Register C eingestellt, Quelladresse ueber Register HL. B arbeitet als Bytezaehler. B wird dekrementiert, HL inkrementiert. Es wird eine Blockuebertragung durchgefuehrt, bis B = Ø ist.<br>(C) <--- (HL)<br>B <--- B-1<br>HL <--- HL+1<br>Wiederholen bis B = Ø |
| --- | OUTD | Kanaladresse wird indirekt ueber Register C eingestellt, Quelladresse ueber Register HL. B kann als Bytezaehler arbeiten. B und HL werden dekrementiert.<br>(C) <--- (HL)<br>B <--- B-1<br>HL <--- HL-1 |
| --- | OTDR | Kanaladresse wird indirekt ueber das Register C eingestellt, Quelladresse ueber Register HL. B arbeitet als Bytezaehler. B und HL werden dekrementiert. Es wird eine Blockuebertragung durchgefuehrt, bis B=Ø ist. |

| 8080-Mnemonik | Z80-Mnemonik | Wirkungsweise der Befehle |
|---|---|---|
| | | (C) <--- (HL)<br>B <--- B-1<br>HL <--- HL-1<br>Wiederholen bis B=Ø ist. |

## 1.6.16. Abkuerzungsverzeichnis zur Befehlsbeschreibung

```
r    : eines der Register  A, B, C, D, E, H oder L
r'   : eines der Register  A', B', C', D', E', H' oder L'
dd   : eines der Doppelregister BC, DE, HL oder SP
qq   : eines der Doppelregister AF, BC, DE oder HL
pp   : eines der Doppelregister BC, DE oder SP
bb   : eines der Doppelregister BC, DE, HL, SP, IX oder IY
s    : r, n, (HL), (IX+d) und (IY+d) sind erlaubt.
t    : r, (HL), (IX+d) und (IY+d) sind moeglich.
n    : 8-Bit-Direktoperand
nn   : 16-Bit-Direktoperand
d    : Verschiebung bei Adressierung ueber Indexregister,
       erlaubt im Bereich von -128 ≤ d ≤ 127
       Die Bereichsgrenzen werden vom Assembler nicht geprueft!
e    : relative Sprungsadresse,
       erlaubt im Bereich von -128 ≤ e ≤ 127
       Die Bereichsgrenzen werden vom Assembler nicht geprueft!
b    : Bit, das in den Bitmanipulierbefehlen behandelt werden
       soll Ø ≤ b ≤ 7
M    : Inhalt des durch HL adressierten Speicherplatzes
k    : Die Werte Ø, 1, 2, 3, 4, 5, 6, 7 sind erlaubt.
p    : Die  Werte ØØH,  Ø8H,  1ØH,  18H,  28H,  3ØH,  38H sind
       erlaubt.
CY   : Carry-Flag
```

**Anmerkung:**
In der 8080-Mnemonik wird fuer die Operanden nur der erste Buchstabe geschrieben:

- . H fuer HL
- . D fuer DE
- . B fuer BC.

Aber es bleibt SP, und fuer AF wird PSW verwendet. Fuer M darf nicht (HL) geschrieben werden.

## 6.6.17. Arbeit mit den Bedingungsbits (Flags)

Das Flagregister F gibt Auskunft ueber das Ergebnis der letzten Prozessoroperation.
Es dient im wesentlichen dazu, bedingte Programmverzweigungen bzw. bedingte Unterprogrammaufrufe oder -ruecksprueng auszufuehren.

**Flagregister:**

| | | |
|---|---|---|
| S | - Vorzeichenbit | (Sign-Flag) |
| Z | - Nullbit | (Zero-Flag) |
| H | - Halbbyteueberlaufbit | (Half-Carry-Flag) |
| N | - Additions- / Subtraktionsbit | |
| P/V | - Paritaets- / Uebertragsbit | (Parity/Overflow-Flag) |
| CY | - Uebertragsbit | (Carry-Flag) |
| X | - nicht belegt | |

**Vorzeichenbit (S-Flag):**

Bei bestimmten Befehlen wird das hoechstwertige Bit des Akkumulators geladen.
Bei Ausfuehrung von arithmetischen Befehlen mit vorzeichenbehafteten Zahlen wird eine positive Zahl durch eine Ø und eine negative Zahl durch eine 1 in der hoechstwertigen Bitstelle gekennzeichnet.

**Nullbit (Z-Flag):**

Es wird bei arithmetischen und logischen 1-Byte-Operationen gesetzt, wenn das Ergebnisbyte des Akkumulators Ø ist. Sonst wird das Ergebnisbyte zurueckgesetzt.

Bei Vergleichs- und Suchbefehlen wird das Z-Flag gesetzt, sobald der Vergleich positiv ausfaellt.

Bei den Bit-Befehlen wird das Z-Flag mit dem komplementaeren Wert des getesteten Bits geladen.

Bei der Uebertragung eines Bytes zwischen einer Speicherstelle und einer E/A-Schnittstelle (INI, IND, OUTI, OUTD) wird das Z-Flag 1 gesetzt, wenn der Wert des Zaehlregisters Ø wird.
Bei IN r,(C) wird das Z-Flag gesetzt, wenn die eingezogenen bzw. am E/A-Tor anliegenden Daten den Wert Ø haben.

**Halbbyte-Uebertragsbit (H-Flag):**

Das H-Flag wird entsprechend dem Uebertragungsergebnis zwischen den Bits 3 und 4 einer arithmetischen 1-Byte-Operation gesetzt

(falls Uebertrag) oder rueckgesetzt (falls kein Uebertrag). Es wird beim Befehl DAA verwendet, um das Ergebnis einer gepackten BCD-Addition bzw. -Subtraktion zu korrigieren.

| H | Addition | Subtraktion |
|---|---|---|
| 1 | Uebertrag von Bit 3 zu Bit 4 | Negativer Uebertrag von Bit 4 |
| Ø | Kein Uebertrag von Bit 3 zu Bit 4 | Kein negativer Uebertrag von Bit 4 |

**Paritaets- / Ueberlaufbit (P/V-Flag):**

Es wird unterschiedlich genutzt.
Bei arithmetischen Befehlen wird das P/V-Flag gesetzt, wenn im Ergebnis das hoechste Bit des Akkumulators gesetzt wird.

Bei logischen Operationen und Verschiebebefehlen dient das P/V-Flag zur Ueberpruefung der Paritaet des Ergebnisses. Ist die Anzahl der gesetzten Bits im angesprochenen Byte gerade (Ø, 2, 4, 6, 8), so wird das P/V-Flag gesetzt, ansonsten rueckgesetzt.

Bei Blocktransport- (LDI, LDIR, LDD, LDDR) und Blocksuchbefehlen (CPI, CPIR, CPD, CPDR) gibt das P/V-Flag Auskunft ueber den Stand des Bytezaehlers. Das P/V-Flag wird rueckgesetzt, wenn nach Dekrementieren des Bytezaehlers (=<BC>) als Ergebnis Ø entsteht. In allen uebrigen Faellen bleibt das P/V-Flag 1.

**Additions- / Subtraktionsbit (N-Flag):**

Es wird intern bei dem DAA-Befehl benutzt, um zwischen Additions- und Subtraktionsbefehlen zu unterscheiden. Bei allen Additionsbefehlen wird das N-Flag rueckgesetzt. Subtraktionsbefehle setzen das N-Flag.

**Uebertragsbit (CY-Flag):**

Das Setzen / Ruecksetzen des CY-Flags wird je nach ausgefuehrter Operation verschieden behandelt.

Das CY-Flag wird gesetzt, wenn
- bei Additionsbefehlen ein Uebertrag entsteht und
- bei Subtraktionsbefehlen ein negativer Uebertrag entsteht.

Es wird rueckgesetzt,
- bei Additionsbefehlen, die keinen Uebertrag erzeugen und
- bei Subtraktionsbefehlen, die keinen negativen Uebertrag erzeugen.

Bei Verschiebebefehlen (RLA, RLC, RL, RR) wird das CY-Flag als Zwischenspeicher fuer die Uebertragung des niederwertigsten bzw. hoechstwertigen Bits eines CPU-Registers bzw. Speicherplatzes benutzt.

Bei den Befehlen RLCA, RLC und SLA enthaelt das CY-Flag den Wert des hoechstwertigen Bits, das durch den Befehl aus dem behandelten Register bzw. Speicherplatz hinausgeschoben wurde.

Bei den Befehlen RRCA, RRC, SRA, SRL enthaelt das CY-Flag analog den Wert des niederwertigsten Bits.

Die logischen Befehle AND, OR, XOR setzen das CY-Flag grundsaetzlich zurueck.

Die speziell fuer das CY-Flag vorgesehenen Befehle SCF (Setzen CY-Flag) und CCF (Komplement C-Flag) erlauben das Setzen bzw. Komplementieren des CY-Flags.

## 1.7. Cross-Referenz

Das Cross-Referenz-Programm verarbeitet eine speziell uebersetzte Druckdatei zu einer Liste aller Adressverweise innerhalb des Moduls einschliesslich der Speicherplaetze, wo sie definiert sind.
Die Cross-Referenz kann als Hilfsmittel beim Programmtest genutzt werden.
Das Cross-Referenz-Programm erlaubt dem Programmierer das Verarbeiten einer vom Assembler erzeugten Cross-Referenz-Datei.
Diese Datei enthaelt Steuerzeichen, die waehrend der Uebersetzung mit dem Assembler eingesetzt wurden.

Das Cross-Referenz-Programm erzeugt eine Liste, die der .PRN-Datei gleicht und zwei zusaetzliche Merkmale besitzt:

1. Jede Quellanweisung wird mit einer Cross-Referenz-Nummer versehen.

2. Am Listenende erscheinen die Variablennamen in alphabetischer Reihenfolge.
   Jedem Namen werden die Nummer der Zeilen in aufsteigender Folge zugeordnet, wo er auftritt. Die Nummer der Zeile, wo er definiert wurde, ist durch # gekennzeichnet.

Die Cross-Referenz-Liste ersetzt die .PRN-Datei des Assemblers und erhaelt auch den Dateityp .PRN.

### 1.7.1. Erzeugen einer Cross-Referenz-Liste

Das Erzeugen einer Cross-Referenz-Liste erfolgt in zwei Schritten:

1. Erzeugen einer Cross-Referenz-Datei (.CRF)
2. Generieren einer Cross-Referenz-Liste (.PRN)

Der erste Schritt wird im Assembler ausgefuehrt; der zweite im Cross-Referenz-Programm.

#### 1.7.1.1. Erzeugen einer Cross-Referenz-Datei

Fuer das Erzeugen einer Cross-Referenz-Datei muss in der Kommandozeile des Assemblers der /C-Schalter gesetzt werden.

Beispiel: ASM =PROG/C

Damit wird PROG.MAC uebersetzt und PROG.REL (Objektdatei) und PROG.CRF (Cross-Referenz-Datei) erzeugt.

#### 1.7.1.2. Generieren der Cross-Referenz-Liste

Die Cross-Referenz-Liste wird beim Durchlaufen der .CRF-Datei durch das Cross-Referenz-Programm erzeugt.

Der Aufruf erfolgt mit REF

Das Programm meldet sich mit einem Stern (*). Nun ist der Name der .CRF-Datei einzugeben:

=dateiname

Beispiel: REF =PROG

Es wird die Datei PROG.PRN erzeugt.
Diese .PRN-Datei kann unter Nutzung der Betriebssystemkommandos gedruckt oder auf dem Bildschirm angezeigt werden.
Zusaetzlich unterstuetzt REF die gleichen Ausgabegeraete wie der Assembler.

Beispiel: REF LST:=PROG

gibt die Liste auf den Drucker aus. Es wird keine Diskettendatei erzeugt.

REF TTY:=PROG

Die Ausgabe erfolgt nur auf dem Bildschirm.
Es ist auch moeglich, ein anderes Laufwerk fuer die Ausgabe anzugeben.

Beispiel: REF B:=A:PROG

Die Diskette mit PROG.CRF befindet sich im Laufwerk A; PROG.PRN

soll auf die Diskette gebracht werden, die im Laufwerk B liegt.

Nach Beenden meldet sich REF mit einem Stern (*) zurueck, es kann ein neuer Dateiname eingegeben werden.

Die Rueckkehr zum Betriebssytem erfolgt mittels ^C.

Die Druckdatei kann auch einen anderen Namen und einen anderen Dateityp erhalten, dann sind diese anzugeben.

Beispiel:

```
     REF  PROG.CRL=PROG
bzw.
     REF  PROGREF=PROG
```

Die obere Kommandozeile erzeugt eine Druckdatei mit dem Namen PROG.CRL; die untere generiert eine Datei mit der Bezeichnung PROGREF.PRN.
Dateitypen koennen angegeben werden, um die Cross-Referenz-Liste von der Druckdatei des Assemblers zu unterscheiden.

### 1.7.2. Pseudooperationen zur Listensteuerung

Die Option fuer das Erzeugen einer Cross-Referenz kann fuer Programmabschnitte aber nicht fuer das ganze Programm angewendet werden.
Zum Steuern des Auflistens oder Weglassens der Cross-Referenz sind in der Quelldatei die Pseudooperationen .CREF und .XCREF anzugeben. Sie koennen an jeder beliebigen Stelle im Programm in das Operationsfeld geschrieben werden.
_.CREF und .XCREF haben keine Argumente.

**.CREF** Erzeugen der Cross-Referenz

_.CREF ist die Standardbedingung: .CREF ist zu verwenden, wenn nach einem .XCREF das Erzeugen der Cross-Referenz wieder gestattet werden soll.
.CREF wirkt, bis ein .XCREF erscheint.

Bemerkung:
_.CREF wirkt nur, wenn in der Kommandozeile des Assemblers der /C-Schalter gesetzt wurde.

**.XCREF** Unterdruecken der Cross-Referenz

_.XCREF schaltet die Wirkung von .CREF aus. .XCREF unterdrueckt die Cross-Referenz fuer Programmabschnitte.
Weil weder .CREF noch .XCREF Wirkung haben, wenn nicht der /C-Schalter in der Assembler-Kommandozeile gesetzt ist muss .XCREF nicht verwendet werden, wenn eine normale Liste (ohne Cross-Referenz) gewuenscht wird. Das wird auch erreicht einfach durch Weglassen des /C-Schalters in der Kommandozeile.

## 1.8. Uebersicht ueber die Pseudooperationen des Assemblers

### 1.8.1. Einzelfunktions-Pseudooperationen

**Pseudooperationen zur Auswahl der Anweisungsliste**


| | | Seite |
|---|---|---|
| | .Z80 | 23 |
| | .8080 | 23 |


**Pseudooperationen zur Daten- und Symboldefinition**


| | | Seite |
|---|---|---|
| <name> | ASET <exp> | 27 |
| | BYTE EXT <symbol> | 26 |
| | BYTE EXTRN <symbol> | 26 |
| | BYTE EXTERNAL <symbol> | 26 |
| | DB <exp>[,<exp>...] | 23 |
| | DC <string>[,<string>...] | 24 |
| | DEFB <exp>[,<exp>...] | 23 |
| <name> | DEFL <exp> | 27 |
| | DEFM <exp>[,<exp>...] | 23 |
| | DEFS <exp1>[,<exp2>] | 24 |
| | DEFW <exp>[,<exp>...] | 25 |
| | DS <exp1>[,<exp2>] | 24 |
| | DW <exp>[,<exp>...] | 25 |
| | ENTRY <name>[,<name>...] | 26 |
| <name> | EQU <exp> | 25 |
| | EXT <name>,[<name>...] | 26 |
| | EXTRN <name>[,<name>...] | 26 |
| | EXTERNAL <name>[,<name>...] | 26 |
| | GLOBAL <name>[,<name>...] | 26 |
| | PUBLIC <name>[,<name>...] | 26 |
| <name> | SET <exp> nur fuer 8080 | 27 |


**Pseudooperationen zur Zuweisung des Speicherzuordnungszaehlers**


| | Seite |
|---|---|
| ASEG | 28 |
| CSEG | 28 |
| DSEG | 29 |
| COMMON /[<blockname>]/ | 30 |
| ORG <exp> | 31 |
| .PHASE <exp> | 31 |
| .DEPHASE | 31 |

**Dateibezogene Pseudooperationen**


| | Seite |
|---|---|
| .COMMENT <delim><text><delim> | 32 |
| END [<exp>] | 33 |
| INCLUDE <dateiname> | 33 |
| $INCLUDE <dateiname> | 33 |
| MACLIB <dateiname> | 33 |
| NAME ('<modulname>') | 34 |
| .RADIX <exp> | 34 |
| .REQUEST <dateiname>[,<dateiname>...] | 35 |


## 1.8.2. Pseudooperationen zur Listensteuerung

**Pseudooperationen zur Formatsteuerung**


| | Seite |
|---|---|
| *EJECT [<exp>] | 36 |
| $EJECT [<exp>] | 36 |
| PAGE <exp> | 36 |
| SUBTTL [<text>] | 37 |
| TITLE <text> | 36 |


**Allgemeine Pseudooperationen zur Listensteuerung**


| | Seite |
|---|---|
| .LIST | 37 |
| .XLIST | 37 |
| .PRINTX <delim><text><delim> | 38 |


**Pseudooperationen zur bedingten Listensteuerung**


| | Seite |
|---|---|
| .SFCOND | 38 |
| .LFCOND | 38 |
| .TFCOND | 38 |


**Pseudooperationen zur Listensteuerung der Makroerweiterung**


| | Seite |
|---|---|
| .LALL | 40 |
| .SALL | 40 |
| .XALL | 40 |


**Pseudooperationen zur Steuerung der Cross-Referenz**


| | Seite |
|---|---|
| .XCREF | 41 |
| .CREF | 41 |

## 1.8.3. Pseudooperationen zur Makrofaehigkeit

**Pseudooperationen zur Makrodefinition**

| | | Seite |
|---|---|---|
| <name> | MACRO <parameter>[,<parameter>...] | 42 |
| | ENDM | 47 |
| | EXITM | 47 |
| | LOCAL <parameter>[,<parameter>...] | 48 |

**Wiederholungspseudooperationen**

| | Seite |
|---|---|
| REPT <exp> | 47 |
| IRP <dummy>,<parameter> | 45 |
| IRPC <dummy>,<string> | 46 |

**Pseudooperationen zur bedingten Assemblierung**

| | Seite |
|---|---|
| COND <exp> | 51 |
| ELSE | 51 |
| ENDC | 52 |
| ENDIF | 52 |
| IF <exp> | 51 |
| IFB <arg> | 51 |
| IFDEF <symbol> | 51 |
| IFDIF <<arg1>>,<<arg2>> | 51 |
| IFE <exp> | 51 |
| IFF <exp> | 51 |
| IFIDN <<arg1>>,<<arg2>> | 51 |
| IFNB <arg> | 51 |
| IFNDEF <symbol> | 51 |
| IFT <exp> | 51 |
| IF1 | 51 |
| IF2 | 51 |

## 1.9. Mikrobefehlsliste - Z8Ø-Operationskodes

Mikrobefehlsliste - Z8Ø-Operationskodes — Blatt 1

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|---|
| 8-Bit-Ladebefehle | | | | | | |
| LD r,r' | r <-- r' | | r/r' \| A B C D E H L<br>A \|7F 78 79 7A 7B 7C 7D<br>B \|47 4Ø 41 42 43 44 45<br>C \|4F 48 49 4A 4B 4C 4D<br>D \|57 5Ø 51 52 53 54 55<br>E \|5F 58 59 5A 5B 5C 5D<br>H \|67 6Ø 61 62 63 64 65<br>L \|6F 68 69 6A 6B 6C 6D | | | |
| LD r,n | r <-- n | | r \| A B C D E H L<br>\|3E Ø6 ØE 16 1E 26 2E | n | | ..... |
| LD r,(HL) | r <-- (HL) | | r \| A B C D E H L<br>\|7E 46 4E 56 5E 66 6E | | | ..... |
| LD r,(IX+d) | r <-- (IX+d) | DD | r \| A B C D E H L<br>\|7E 46 4E 56 5E 66 6E | d | | ..... |

Mikrobefehlsliste - Z8Ø-Operationskodes Blatt 2

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| LD r,(IY+d) | r <-- (IY+d) | | r \| A B C D E H L | | | |
| | | FD | 7E 46 4E 56 5E 66 6E | d | | ..... |
| LD (HL),r | (HL) <-- r | | r \| A B C D E H L | | | |
| | | | 77 7Ø 71 72 73 74 75 | | | ..... |
| LD (IX+d),r | (IX+d)<-- r | | r \| A B C D E H L | | | |
| | | DD | 77 7Ø 71 72 73 74 75 | d | | ..... |
| LD (IY+d),r | (IY+d)<-- r | | r \| A B C D E H L | | | |
| | | FD | 77 7Ø 71 72 73 74 75 | d | | ..... |
| LD (HL),n | (HL) <-- n | | 36 | n | | ..... |
| LD (IX+d),n | (IX+d)<-- n | DD | 36 | d | n | ..... |
| LD (IY+d),n | (IY+d)<-- n | FD | 36 | d | n | ..... |
| LD A,(BC) | A <-- (BC) | | ØA | | | ..... |
| LD A,(DE) | A <-- (DE) | | 1A | | | ..... |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| LD (BC),A | (BC) <-- A | | Ø2 | | | ..... |
| LD (DE),A | (DE) <-- A | | 12 | | | ..... |
| LD A,(nn) | A <-- (nn) | | 3A | $n_L$ | $n_H$ | ..... |
| LD (nn),A | (nn) <-- A | | 32 | $n_L$ | $n_H$ | ..... |
| LD A,I | A <-- I | ED | 57 | | | .\|\|\|ØØ |
| LD I,A | I <-- A | ED | 47 | | | ..... |
| 16-Bit-Ladebefehle | | | | | | |
| LD dd,nn | dd <-- nn | | dd \| BC DE HL SP<br>---\|---<br>Ø1 11 21 31 | $n_L$ | $n_H$ | ..... |
| LD IX,nn | IX <-- nn | DD | 21 | $n_L$ | $n_H$ | ..... |
| LD IY,nn | IY <-- nn | FD | 21 | $n_L$ | $n_H$ | ..... |
| LD dd,(nn) | $dd_H$ <-- (nn+1) | | dd \| BC DE HL SP<br>---\|---<br>4B 5B 6B 7B | | | |
| | $dd_L$ <-- (nn) | ED | | $n_L$ | $n_H$ | ..... |

Mikrobefehlsliste - Z8Ø-Operationskodes Blatt 4

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|---|
| LD HL,(nn) | H <-- (nn+1)<br>L <-- (nn) | | 2A | $n_L$ | $n_H$ | ..... |
| LD IX,(nn) | $IX_H$ <-- (nn+1)<br>$IX_L$ <-- (nn) | DD | 2A | $n_L$ | $n_H$ | ..... |
| LD IY,(nn) | $IY_H$ <-- (nn+1)<br>$IY_L$ <-- (nn) | FD | 2A | $n_L$ | $n_H$ | ..... |
| LD (nn),dd | (nn+1)<-- $dd_H$<br>(nn) <-- $dd_L$ | ED | dd \| BC DE HL SP<br>---- 43 53 63 73 | $n_L$ | $n_H$ | ..... |
| LD (nn),HL | (nn+1)<-- H<br>(nn) <-- L | | 22 | $n_L$ | $n_H$ | ..... |
| LD (nn),IX | (nn+1)<-- $IX_H$<br>(nn) <-- $IX_L$ | DD | 22 | $n_L$ | $n_H$ | ..... |

Mikrobefehlsliste - Z8Ø-Operationskodes Blatt 5

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| LD (nn),IY | (nn+1)<-- $IY_H$ | FD | 22 | $n_L$ | $n_H$ | ..... |
| | (nn) <-- $IY_L$ | | | | | |
| LD SP,HL | SP <-- HL | | F9 | | | ..... |
| LD SP,IX | SP <-- IX | DD | F9 | | | ..... |
| LD SP,IY | SP <-- IY | FD | F9 | | | ..... |
| PUSH dd | (SP-2)<-- $dd_L$ | | dd \| BC DE HL AF | | | |
| | (SP-1)<-- $dd_H$ | | \| C5 D5 E5 F5 | | | ..... |
| PUSH IX | (SP-2)<-- $IX_L$ | DD | E5 | | | ..... |
| | (SP-1)<-- $IX_H$ | | | | | |
| PUSH IY | (SP-2)<-- $IY_L$ | FD | E5 | | | ..... |
| | (SP-1)<-- $IY_H$ | | | | | |

Mikrobefehlsliste - Z8ø-Operationskodes Blatt 6

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| POP dd | $dd_H$ <-- (SP+1) | | dd \| BC DE HL AF | | | |
| | $dd_L$ <-- (SP) | | \| C1 D1 E1 F1 | | | ..... |
| POP IX | $IX_H$ <-- (SP+1) | DD | E1 | | | ..... |
| | $IX_L$ <-- (SP) | | | | | |
| POP IY | $IY_H$ <-- (SP+1) | FD | E1 | | | ..... |
| | $IY_L$ <-- (SP) | | | | | |
| Austauschbefehle | | | | | | |
| EX DE,HL | DE <--> HL | | EB | | | ..... |
| EX (SP),HL | H <--> (SP+1)<br>L <--> (SP) | | E3 | | | ..... |
| EX (SP),IX | $IX_H$ <--> (SP+1) | DD | E3 | | | ..... |
| | $IX_L$ <--> (SP) | | | | | |

Mikrobefehlsliste - Z8Ø-Operationskodes Blatt 7

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| EX (SP),IY | $IY_H$ <--> (SP+1) | FD | E3 | | | ..... |
| | $IY_L$ <--> (SP) | | | | | |
| EX AF,AF' | AF <--> AF' | | Ø8 | | | ..... |
| EXX | BC <--> BC' | | | | | |
| | DE <--> DE' | | D9 | | | ..... |
| | HL <--> HL' | | | | | |
| Blocktransportbefehle | | | | | | |
| LDI | (DE) <-- (HL) | ED | AØ | | | ..Ø\|Ø. |
| | DE <-- DE+1 | | | | | a |
| | HL <-- HL+1 | | | | | |
| | BC <-- BC-1 | | | | | |
| LDIR | wie LDI | ED | BØ | | | ..ØØØ. |
| | bis BC = Ø | | | | | |
| LDD | (DE) <-- (HL) | ED | A8 | | | ..Ø\|Ø. |
| | DE <-- DE-1 | | | | | a |
| | HL <-- HL-1 | | | | | |
| | BC <-- BC-1 | | | | | |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| LDDR | wie LDD<br>bis BC = Ø | ED | B8 | | | ..ØØØ. |
| Suchbefehle | | | | | | |
| CPI | A - (HL)<br>HL <-- HL+1<br>BC <-- BC-1 | ED | A1 | | | ````1.<br>b a |
| CPIR | wie CPI<br>bis BC = Ø<br>oder A = (HL) | ED | B1 | | | ````1.<br>b a |
| CPD | A - (HL)<br>HL <-- HL-1<br>BC <-- BC-1 | ED | A9 | | | ````1.<br>b a |
| CPDR | wie CPD<br>bis BC = Ø<br>oder A = (HL) | ED | B9 | | | ````1.<br>b a |
| 8-Bit-Arithmetik-Befehle | | | | | | |
| ADD A,r | A <-- A+r | | r \| A B C D E H L<br>---\|---<br>\|87 8Ø 81 82 83 84 85 | | | ```V Ø\| |

Mikrobefehlsliste - Z8Ø-Operationskodes Blatt 9

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| ADC A,r | A <-- A+r+CY | | r \| A B C D E H L<br>----\|----------<br>\|8F 88 89 8A 8B 8C 8D | | | \|\|\|V\|Ø\|\| |
| ADD A,n | A <-- A+n | | C6 | n | | \|\|\|V\|Ø\|\| |
| ADC A,n | A <-- A+n+CY | | CE | n | | \|\|\|V\|Ø\|\| |
| ADD A,(HL) | A <-- A+(HL) | | 86 | | | \|\|\|V\|Ø\|\| |
| ADD A,(IX+d) | A <-- A+(IX+d) | DD | 86 | d | | \|\|\|V\|Ø\|\| |
| ADD A,(IY+d) | A <-- A+(IY+d) | FD | 86 | d | | \|\|\|V\|Ø\|\| |
| ADC A,(HL) | A <-- A+(HL)+CY | | 8E | | | \|\|\|V\|Ø\|\| |
| ADC A,(IX+d) | A <-- A+(IX+d)+CY | DD | 8E | d | | \|\|\|V\|Ø\|\| |
| ADC A,(IY+d) | A <-- A+(IY+d)+CY | FD | 8E | d | | \|\|\|V\|Ø\|\| |
| SUB r | A <-- A-r | | r \| A B C D E H L<br>----\|----------<br>\|97 9Ø 91 92 93 94 95 | | | \|\|\|V\|1\|\| |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| SBC A,r | A <-- A-r-CY | | r \| A B C D E H L<br>----\|----------<br>\|9F 98 99 9A 9B 9C 9D | | | \|\|\|V1\| |
| SUB n | A <-- A-n | | D6 | n | | \|\|\|V1\| |
| SBC A,n | A <-- A-n-CY | | DE | n | | \|\|\|V1\| |
| SUB (HL) | A <-- A-(HL) | | 96 | | | \|\|\|V1\| |
| SUB (IX+d) | A <-- A-(IX+d) | DD | 96 | d | | \|\|\|V1\| |
| SUB (IY+d) | A <-- A-(IY+d) | FD | 96 | d | | \|\|\|V1\| |
| SBC A,(HL) | A <-- A-(HL)-CY | | 9E | | | \|\|\|V1\| |
| SBC A,(IX+d) | A <-- A-(IX+d)-CY | DD | 9E | d | | \|\|\|V1\| |
| SBC A,(IY+d) | A <-- A-(IY+d)-CY | FD | 9E | d | | \|\|\|V1\| |
| CP r | A-r | | r \| A B C D E H L<br>----\|----------<br>\|BF B8 B9 BA BB BC BD | | | \|\|\|V1\| |
| CP n | A-n | | FE | n | | \|\|\|V1\| |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|---|
| CP (HL) | A-(HL) | | BE | | | \|\|\|V1\|\| |
| CP (IX+d) | A-(IX+d) | DD | BE | d | | \|\|\|V1\|\| |
| CP (IY+d) | A-(IY+d) | FD | BE | d | | \|\|\|V1\|\| |
| INC r | r <-- r+1 | | r \| A B C D E H L<br>----\|----------<br>3C Ø4 ØC 14 1C 24 2C | | | \|\|\|VØ.\| |
| INC (HL) | (HL) <-- (HL)+1 | | 34 | | | \|\|\|VØ.\| |
| INC (IX+d) | (IX+d)<-- (IX+d)+1 | DD | 34 | d | | \|\|\|VØ.\| |
| INC (IY+d) | (IY+d)<-- (IY+d)+1 | FD | 34 | d | | \|\|\|VØ.\| |
| DEC r | r <-- r-1 | | r \| A B C D E H L<br>----\|----------<br>3D Ø5 ØD 15 1D 25 2D | | | \|\|\|V1.\| |
| DEC (HL) | (HL) <-- (HL)-1 | | 35 | | | \|\|\|V1.\| |
| DEC (IX+d) | (IX+d)<-- (IX+d)-1 | DD | 35 | d | | \|\|\|V1.\| |
| DEC (IY+d) | (IY+d)<-- (IY+d)-1 | FD | 35 | d | | \|\|\|V1.\| |

..

Mikrobefehlsliste - Z8Ø-Operationskodes Blatt 12

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| Allgemeine_Arithmetik_-und_Steuerbefehle | | | | | | |
| DAA | Dezimalkorrektur nach Operation mit gepackten Zahlen | | 27 | | | \|\|\|P.\| |
| CPL | A <-- $\overline{A}$ | | 2F | | | ..1.1. |
| NEG | A <-- $\overline{A}$+1 | ED | 44 | | | \|\|\|V1\| |
| CCF | CY <-- $\overline{CY}$ | | 3F | | | ..X.Ø\| |
| SCF | CY <-- 1 | | 37 | | | ..Ø.Ø1 |
| NOP | keine Operation | | ØØ | | | ..... |
| HALT | CPU im Halt-Zustand | | 76 | | | ..... |
| DI | IFF <-- Ø | | F3 | | | ..... |
| EI | IFF <-- 1 | | FB | | | ..... |
| IM Ø | Setzen des Interrupt-Mode Ø | ED | 86 | | | ..... |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| IM 1 | Setzen des Interrupt-Mode 1 | ED | 96 | | | ..... |
| IM 2 | Setzen des Interrupt-Mode 2 | ED | 9E | | | ..... |
| 8-Bit-Logik-Befehle | | | | | | |
| A\|r\|&\|v\|⊕<br>Ø\|Ø\|Ø\|Ø\|Ø<br>Ø\|1\|Ø\|1\|1<br>1\|Ø\|Ø\|1\|1<br>1\|1\|1\|1\|Ø | | | | | | |
| AND r | A <-- A&r | | r \| A B C D E H L<br>----\|----------<br>\|A7 AØ A1 A2 A3 A4 A5 | | | \|\|1PØØ |
| AND n | A <-- A&n | | E6 | n | | \|\|1PØØ |
| AND (HL) | A <-- A&(HL) | | A6 | | | \|\|1PØØ |
| AND (IX+d) | A <-- A&(IX+d) | DD | A6 | d | | \|\|1PØØ |
| AND (IY+d) | A <-- A&(IY+d) | FD | A6 | d | | \|\|1PØØ |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|---|
| OR r | A <-- Avr | | r \| A B C D E H L<br>B7 BØ B1 B2 B3 B4 B5 | | | \|\|ØPØØ |
| OR n | A <-- Avn | | F6 | n | | \|\|ØPØØ |
| OR (HL) | A <-- Av(HL) | | B6 | | | \|\|ØPØØ |
| OR (IX+d) | A <-- Av(IX+d) | DD | B6 | d | | \|\|ØPØØ |
| OR (IY+d) | A <-- Av(IY+d) | FD | B6 | d | | \|\|ØPØØ |
| XOR r | A <-- A⊕r | | r \| A B C D E H L<br>AF A8 A9 AA AB AC AD | | | \|\|ØPØØ |
| XOR n | A <-- A⊕n | | EE | n | | \|\|ØPØØ |
| XOR (HL) | A <-- A⊕(HL) | | AE | | | \|\|ØPØØ |
| XOR (IX+d) | A <-- A⊕(IX+d) | DD | AE | d | | \|\|ØPØØ |
| XOR (IY+d) | A <-- A⊕(IY+d) | FD | AE | d | | \|\|ØPØØ |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|---|
| 16-Bit-Arithmetik-Befehle | | | | | | |
| ADD HL,dd | HL <-- HL+dd | | dd \| BC DE HL SP<br>---\|---------- | | | |
| | | | Ø9 19 29 39 | | | ..X.Ø\| |
| ADD IX,dd | IX <-- IX+dd | | dd \| BC DE IX SP<br>---\|---------- | | | |
| | | DD | Ø9 19 29 39 | | | ..X.Ø\| |
| ADD IY,dd | IY <-- IY+dd | | dd \| BC DE IY SP<br>---\|---------- | | | |
| | | FD | Ø9 19 29 39 | | | ..X.Ø\| |
| ADC HL,dd | HL <-- HL+dd+CY | | dd \| BC DE HL SP<br>---\|---------- | | | |
| | | ED | 4A 5A 6A 7A | | | \|\|XVØ\| |
| SBC HL,dd | HL <-- HL-dd-CY | | dd \| BC DE HL SP<br>---\|---------- | | | |
| | | ED | 42 52 62 72 | | | \|\|XVØ\| |
| INC dd | dd <-- dd+1 | | dd \| BC DE HL SP<br>---\|---------- | | | |
| | | | Ø3 13 23 33 | | | ..... |
| INC IX | IX <-- IX+1 | DD | 23 | | | ..... |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| INC IY | IY <-- IY+1 | FD | 23 | | | ..... |
| DEC dd | dd <-- dd-1 | | dd \| BC DE HL SP<br>----\|----------<br>\| ØB 1B 2B 3B | | | ..... |
| DEC IX | IX <-- IX-1 | DD | 2B | | | ..... |
| DEC IY | IY <-- IY-1 | FD | 2B | | | ..... |
| Rotations-_und_Verschiebebefehle | | | | | | |
| RLCA | | | Ø7 | | | ..Ø.Ø. |
| RLA | | | 17 | | | ..Ø.Ø. |
| RRCA | | | ØF | | | ..Ø.Ø. |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|---|
| RRA | | | 1F | | | ..Ø.Ø\| |
| RLC r | | CB | r \| A B C D E H L<br>----\|----------<br>\|Ø7 ØØ Ø1 Ø2 Ø3 Ø4 Ø5 | | | \|\|ØPØ\| |
| RLC (HL) | | CB | Ø6 | | | \|\|ØPØ\| |
| RLC (IX+d) | | DD<br>CB<br>d | Ø6 | | | \|\|ØPØ\| |
| RLC (IY+d) | | FD<br>CB<br>d | Ø6 | | | \|\|ØPØ\| |
| RL r | | CB | r \| A B C D E H L<br>----\|----------<br>\|17 1Ø 11 12 13 14 15 | | | \|\|ØPØ\| |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|---|
| RL (HL) | | CB | 16 | | | \|\|ØPØ\| |
| RL (IX+d) |  | DD<br>CB<br>d | 16 | | | \|\|ØPØ\| |
| RL (IY+d) | | FD<br>CB<br>d | 16 | | | \|\|ØPØ\| |
| RRC r |  | CB | r \| A B C D E H L<br>---\|---<br>\|ØF Ø8 Ø9 ØA ØB ØC ØD | | | \|\|ØPØ\| |
| RRC (HL) |  | CB | ØE | | | \|\|ØPØ\| |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|---|
| RRC (IX+d) |  | DD<br>CB<br>d | ØE | | | \|\|ØPØ\| |
| RRC (IY+d) |  | FD<br>CB<br>d | ØE | | | \|\|ØPØ\| |
| RR r | | <br>CB | r \| A B C D E H L<br>----\|----------<br>\|1F 18 19 1A 1B 1C 1D | | | \|\|ØPØ\| |
| RR (HL) |  | CB | 1E | | | \|\|ØPØ\| |
| RR (IX+d) |  | DD<br>CB<br>d | 1E | | | \|\|ØPØ\| |
| RR (IY+d) | | FD<br>CB<br>d | 1E | | | \|\|ØPØ\| |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| SLA r |  | CB | r \| A B C D E H L<br>----\|--------------------<br>\|27 2Ø 21 22 23 24 25 | | | ‖ØPØ‖ |
| SLA (HL) |  | CB | 26 | | | ‖ØPØ‖ |
| SLA (IX+d) | | DD<br>CB<br>d | 26 | | | ‖ØPØ‖ |
| SLA (IY+d) |  | FD<br>CB<br>d | 26 | | | ‖ØPØ‖ |
| SRA r | | CB | r \| A B C D E H L<br>----\|--------------------<br>\|2F 28 29 2A 2B 2C 2D | | | ‖ØPØ‖ |
| SRA (HL) | | CB | 2E | | | ‖ØPØ‖ |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| SRA (IX+d) | | DD<br>CB | | | | |
| | | | 2E | | | ‖‖ØPØ‖ |
| SRA (IY+d) | | FD<br>CB | | | | |
| | | d | 2E | | | ‖‖ØPØ‖ |
| SRL r | | | r \| A B C D E H L<br>---\|---<br>\|3F 38 39 3A 3B 3C 3D | | | |
| | | CB | | | | ‖‖ØPØ‖ |
| SRL (HL) | | CB | 3E | | | ‖‖ØPØ‖ |
| SRL (IX+d) | | DD<br>CB | | | | |
| | | d | 3E | | | ‖‖ØPØ‖ |
| SRL (IY+d) | | FD<br>CB | | | | |
| | | d | 3E | | | ‖‖ØPØ‖ |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| RLD | | ED | 6F | | | \|\|ØPØ. |
| RRD |  | ED | 67 | | | \|\|ØPØ. |
| BIT b,r | Z <-- $\overline{r_b}$ | CB | b\r \| A B C D E H L<br>Ø \|47 48 41 42 43 44 45<br>1 \|4F 48 49 4A 4B 4C 4D<br>2 \|57 5Ø 51 52 53 54 55<br>3 \|5F 58 59 5A 5B 5C 5D<br>4 \|67 6Ø 61 62 63 64 65<br>5 \|6F 68 69 6A 6B 6C 6D<br>6 \|77 7Ø 71 72 73 74 75<br>7 \|7F 78 79 7A 7B 7C 7D | | | X\|1XØ. |
| BIT b,(HL) | Z <-- $\overline{(HL)_b}$ | CB | b \| Ø 1 2 3 4 5 6 7<br>\|46 4E 56 5E 66 6E 76 7E | | | X\|1XØ. |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|---|
| BIT b,(IX+d) | Z <-- $\overline{(IX+d)_b}$ | DD | b \| Ø 1 2 3 4 5 6 7 | | | X\|1XØ. |
| | | CB | ----\|---------- | | | |
| | | d | 46 4E 56 5E 66 6E 76 7E | | | |
| BIT b,(IY+d) | Z <-- $\overline{(IY+d)_b}$ | FD | b \| Ø 1 2 3 4 5 6 7 | | | X\|1XØ. |
| | | CB | ----\|---------- | | | |
| | | d | 46 4E 56 5E 66 6E 76 7E | | | |
| SET b,r | $r_b$ <-- 1 | | b\r \| A B C D E H L | | | ..... |
| | | | ----\|---------- | | | |
| | | CB | Ø \| C7 C8 C9 CA CB CC CD | | | |
| | | | 1 \| CF C8 C9 CA CB CC CD | | | |
| | | | 2 \| D7 DØ D1 D2 D3 D4 D5 | | | |
| | | | 3 \| DF D8 D9 DA DB DC DD | | | |
| | | | 4 \| E7 EØ E1 E2 E3 E4 E5 | | | |
| | | | 5 \| EF E8 E9 EA EB EC ED | | | |
| | | | 6 \| F7 FØ F1 F2 F3 F4 F5 | | | |
| | | | 7 \| FF F8 F9 FA FB FC FD | | | |
| SET b,(HL) | $(HL)_b$ <-- 1 | | b \| Ø 1 2 3 4 5 6 7 | | | ..... |
| | | | ----\|---------- | | | |
| | | CB | C6 CE D6 DE E6 EE F6 FE | | | |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| SET b,(IX+d) | $(IX+d)_b$ <-- 1 | DD<br>CB<br>d | b \| Ø 1 2 3 4 5 6 7<br>C6 CE D6 DE E6 EE F6 FE | | | ..... |
| SET b,(IY+d) | $(IY+d)_b$ <-- 1 | FD<br>CB<br>d | b \| Ø 1 2 3 4 5 6 7<br>C6 CE D6 DE E6 EE F6 FE | | | ..... |
| RES b,r | $r_b$ <-- Ø | CB | b\r \| A B C D E H L<br>Ø \|87 88 81 82 83 84 85<br>1 \|8F 88 89 8A 8B 8C 8D<br>2 \|97 9Ø 91 92 93 94 95<br>3 \|9F 98 99 9A 9B 9C 9D<br>4 \|A7 AØ A1 A2 A3 A4 A5<br>5 \|AF A8 A9 AA AB AC AD<br>6 \|B7 BØ B1 B2 B3 B4 B5<br>7 \|BF B8 B9 BA BB BC BD | | | ..... |
| RES b,(HL) | $(HL)_b$ <-- Ø | CB | b \| Ø 1 2 3 4 5 6 7<br>86 8E 96 9E A6 AE B6 BE | | | ..... |
| RES b,(IX+d) | $(IX+d)_b$ <-- Ø | DD<br>CB<br>d | b \| Ø 1 2 3 4 5 6 7<br>86 8E 96 9E A6 AE B6 BE | | | ..... |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|---|
| RES b,(IY+d) | $(IY+d)_b$ <-- Ø | FD CB d | b \| Ø 1 2 3 4 5 6 7<br>---\|---<br>\|86 8E 96 9E A6 AE B6 BE | | | ..... |
| Sprungbefehle | | | | | | |
| JP nn | PC <-- nn | | C3 | $n_L$ | $n_H$ | ..... |
| JP cc,nn | PC <-- nn | | cc \|<br>---\|---<br>NZ \| C2<br>Z \| CA<br>NC \| D2<br>C \| DA<br>PO \| E2<br>PE \| EA<br>P \| F2<br>M \| FA | $n_L$ | $n_H$ | ..... |
| | Z = Ø | | | $n_L$ | $n_H$ | ..... |
| | Z = 1 | | | $n_L$ | $n_H$ | ..... |
| | C = Ø | | | $n_L$ | $n_H$ | ..... |
| | C = 1 | | | $n_L$ | $n_H$ | ..... |
| | P = Ø | | | $n_L$ | $n_H$ | ..... |
| | P = 1 | | | $n_L$ | $n_H$ | ..... |
| | S = Ø | | | $n_L$ | $n_H$ | ..... |
| | S = 1 | | | $n_L$ | $n_H$ | ..... |
| JP (HL) | PC <-- HL | | E9 | | | ..... |
| JP (IX) | PC <-- IX | DD | E9 | | | ..... |
| JP (IY) | PC <-- IY | FD | E9 | | | ..... |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| JR e | PC <-- PC+e | | 18 | e-2 | | ..... |
| JR cc,e | PC <-- PC+e | | cc \| ----\|---- NZ \| 2Ø Z \| 28 NC \| 3Ø C \| 38 | | | ..... ..... ..... ..... |
| | Z = Ø<br>Z = 1<br>C = Ø<br>C = 1 | | | | | |
| DJNZ e | PC <-- PC+e, wenn B ≠ Ø | | 1Ø | e-2 | | ..... |
| | B <-- B-1 | | | | | |
| CALL nn | (SP-1)<-- $PC_H$ | | CD | $n_L$ | $n_H$ | ..... |
| | (SP-2)<-- $PC_L$ | | | | | |
| | PC <-- nn | | | | | |
| | SP <-- SP-2 | | | | | |
| CALL cc,nn | (SP-1)<-- $PC_H$ | | | | | |
| | (SP-2)<-- $PC_L$ | | | | | |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|---|
| | PC <-- nn | | | | | |
| | SP <-- SP-2 | | cc<br>----\|---- | | | |
| | Z = Ø | | NZ \| C4 | $n_L$ | $n_H$ | ..... |
| | Z = 1 | | Z \| CC | $n_L$ | $n_H$ | ..... |
| | C = Ø | | NC \| D4 | $n_L$ | $n_H$ | ..... |
| | C = 1 | | C \| DC | $n_L$ | $n_H$ | ..... |
| | P = Ø | | PO \| E4 | $n_L$ | $n_H$ | ..... |
| | P = 1 | | PE \| EC | $n_L$ | $n_H$ | ..... |
| | S = Ø | | P \| F4 | $n_L$ | $n_H$ | ..... |
| | S = 1 | | M \| FC | $n_L$ | $n_H$ | ..... |
| RET cc | $PC_L$ <-- (SP) | | | | | |
| | $PC_H$ <-- (SP+1) | | | | | |
| | SP <-- SP+2 | | cc<br>----\|---- | | | |
| | Z = Ø | | NZ \| CØ | | | ..... |
| | Z = 1 | | Z \| C8 | | | ..... |
| | C = Ø | | NC \| DØ | | | ..... |
| | C = 1 | | C \| D8 | | | ..... |
| | P = Ø | | PO \| EØ | | | ..... |
| | P = 1 | | PE \| E8 | | | ..... |
| | S = Ø | | P \| FØ | | | ..... |
| | S = 1 | | M \| F8 | | | ..... |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| RET | $PC_L$ <-- (SP) | | | | | |
| | $PC_H$ <-- (SP+1) | | C9 | | | ..... |
| | SP <-- SP+2 | | | | | |
| RST p | (SP-1) <-- $PC_H$ | | p \| Ø 1 2 3 4 5 6 7 | | | |
| | (SP-2) <-- $PC_L$ | | C7 CF D7 DF E7 EF F7 FF | | | |
| | $PC_H$ <-- Ø | | | | | |
| | $PC_L$ <-- p | | | | | |
| | SP <-- SP-2 | | | | | |
| Ein_·und_Ausgabebefehle | | | | | | |
| IN A,(n) | A <-- n | | DB | n | | ..... |
| IN r,(C) | r <-- (C) | ED | r \| A B C D E H L<br>78 4Ø 48 5Ø 58 6Ø 68 | | | .\|P\|Ø\| |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| INI | B <-- B-1<br>HL <-- HL+1<br>(HL) <-- (C)<br>B <-- B-1 | ED | A2 | | | .\|XX1X\|<br>c |
| INIR | (HL) <-- (C)<br>B <-- B-1<br>HL <-- HL+1<br>wdh. bis B = Ø | ED | B2 | | | .1XX1X |
| IND | (HL) <-- (C)<br>B <-- B-1<br>HL <-- HL-1 | ED | AA | | | .\|XX1X\|<br>c |
| INDR | (HL) <-- (C)<br>B <-- B-1<br>HL <-- HL-1<br>wdh. bis B = Ø | ED | BA | | | .1XX1X |
| OUT (n),A | n <-- A | | D3 | n | | ..... |
| OUT C,r | (C) <-- r | ED | r \| A B C D E H L<br>79 41 49 51 59 61 69 | | | ..... |

| Befehl | Operation | Vor-byte | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|---|
| OUTI | (C) <-- (HL)<br>B <-- B-1<br>HL <-- HL+1 | ED | A3 | | | .\|XX1X\| c |
| OTIR | (C) <-- (HL)<br>B <-- B-1<br>HL <-- HL+1<br>wdh. bis B = Ø | ED | B3 | | | .1XX1X\| |
| OTDR | (C) <-- (HL)<br>B <-- B-1<br>HL <-- HL-1<br>wdh. bis B = Ø | ED | BB | | | .1XX1X\| |
| OUTD | (C) <-- (HL)<br>B <-- B-1<br>HL <-- HL-1 | ED | AB | | | .\|XX1X\| c |

Zeichenerklaerung:

| | |
|---|---|
| a | P-Flag ist Ø, wenn das Ergebnis von BC-1 = Ø; sonst P = 1 |
| b | Z-Flag ist 1, wenn A = (HL); sonst Z = Ø |
| IFF | Interrupt - Annahme - Flip - Flop |
| c | Wenn das Ergebnis B-1 = Ø, dann wird das Z-Flag gesetzt, sonst ist es geloescht. |

| | |
|---|---|
| . | Flag wird nicht beeinflusst |
| X | Flag unbestimmt |
| \| | Flag entsprechend dem Ergebnis der Operation gesetzt/rueckgesetzt |
| Ø | Flag rueckgesetzt |
| 1 | Flag gesetzt |
| P | gerade Paritaet |
| V | Ueberlauf |
| CY | Carry - Flag |

| | |
|---|---|
| nn | Adresse (2 Byte) |
| $n_L$ | niederwertiges Byte der Adresse nn |
| $n_H$ | hoeherwertiges Byte der Adresse nn |

| | |
|---|---|
| $IX_L$ ($IY_L$, $PC_L$, $dd_L$) | niederwertiger Teil des Doppelregisters |
| $IX_H$ ($IY_H$, $PC_H$, $dd_H$) | hoeherwertiger Teil des Doppelregisters |

## 1.10. Mikrobefehlsliste - 8ø8ø-Operationskodes

Mikrobefehlsliste - 8ø8ø Operationskodes — Blatt 1

| Befehl | Operation | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|
| 8-bit-Ladebefehle | | | | | |
| MOV r,r' | r <-- r' | r/r' \| A B C D E H L<br>A \|7F 78 79 7A 7B 7C 7D<br>B \|47 4ø 41 42 43 44 45<br>C \|4F 48 49 4A 4B 4C 4D<br>D \|57 5ø 51 52 53 54 55<br>E \|5F 58 59 5A 5B 5C 5D<br>H \|67 6ø 61 62 63 64 65<br>L \|6F 68 69 6A 6B 6C 6D | | | ..... |
| MVI r,n | r <-- n | r \| A B C D E H L<br>3E ø6 ØE 16 1E 26 2E | n | | ..... |
| MOV r,M | r <-- (HL) | r \| A B C D E H L<br>7E 46 4E 56 5E 66 6E | | | ..... |
| MOV M,r | (HL) <-- r | r \| A B C D E H L<br>77 7ø 71 72 73 74 75 | | | ..... |
| MVI M,n | (HL) <-- n | 36 | n | | ..... |

Mikrobefehlsliste - 8ØßØ Operationskodes Blatt 2

| Befehl | Operation | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|
| LDAX B | A <-- (BC) | ØA | | | ..... |
| LDAX D | A <-- (DE) | 1A | | | ..... |
| STAX B | (BC) <-- A | Ø2 | | | ..... |
| STAX D | (DE) <-- A | 12 | | | ..... |
| LDA nn | A <-- (nn) | 3A | $n_L$ | $n_H$ | ..... |
| STA nn | (nn) <-- A | 32 | $n_L$ | $n_H$ | ..... |
| 16-bit-Ladebefehle | | | | | |
| LXI d,nn | BC <-- nn<br>DE <-- nn<br>HL <-- nn<br>SP <-- nn | d \| BC DE HL SP<br>----\|----------<br>\| Ø1 11 21 31 | $n_L$ | $n_H$ | ..... |
| LHLD nn | H <-- (nn+1)<br>L <-- (nn) | 2A | $n_L$ | $n_H$ | ..... |
| SHLD nn | (nn+1)<-- H<br>(nn) <-- L | 22 | $n_L$ | $n_H$ | ..... |
| SPHL | SP <-- HL | F9 | | | ..... |

Mikrobefehlsliste - 8ø8ø Operationskodes Blatt 3

| Befehl | Operation | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|
| Indirekte_Registeroperationen | | | | | |
| PUSH PSW | (SP-2)<-- F | F5 | | | ..... |
| | (SP-1)<-- A | | | | |
| PUSH d | (SP-2)<-- $d_L$ | d \| BC DE HL<br>---\|---<br>\| C5 D5 E5 | | | ..... |
| | (SP-1)<-- $d_H$ | | | | |
| POP PSW | A <-- (SP+1)<br>F <-- (SP) | F1 | | | ..... |
| POP d | $d_H$ <-- (SP+1) | d \| BC DE HL<br>---\|---<br>\| C1 D1 E1 | | | ..... |
| | $d_L$ <-- (SP) | | | | |
| Austauschbefehle | | | | | |
| XCHG | DE <--> HL | EB | | | ..... |
| XTHL | H <--> (SP+1)<br>L <--> (SP) | E3 | | | ..... |

Mikrobefehlsliste - 8Ø8Ø Operationskodes Blatt 4

| Befehl | Operation | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|
| 8-bit-Arithmetik | | | | | |
| ADD r | A <-- A+r | r \| A B C D E H L<br>87 8Ø 81 82 83 84 85 | | | \|\|\|VØ\| |
| ADD M | A <-- A+(HL) | 86 | | | \|\|\|VØ\| |
| ADI n | A <-- A+n | C6 | n | | \|\|\|VØ\| |
| ADC r | A <-- A+r+CY | r \| A B C D E H L<br>8F 88 89 8A 8B 8C 8D | | | \|\|\|VØ\| |
| ADC M | A <-- A+(HL)+CY | 8E | | | \|\|\|VØ\| |
| ACI n | A <-- A+n+CY | CE | n | | \|\|\|VØ\| |
| SUB r | A <-- A-r | r \| A B C D E H L<br>97 9Ø 91 92 93 94 95 | | | \|\|\|V1\| |
| SUB M | A <-- A-(HL) | 96 | | | \|\|\|V1\| |
| SUI n | A <-- A-n | D6 | n | | \|\|\|V1\| |

Mikrobefehlsliste - 8Ø8Ø Operationskodes Blatt 5

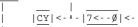
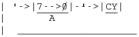
| Befehl | Operation | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|
| SBB r | A <-- A-r-CY | r \| A B C D E H L<br>9F 98 99 9A 9B 9C 9D | | | \|\|\|V1\| |
| SBB M | A <-- A-(HL)-CY | 9E | | | \|\|\|V1\| |
| SBI n | A <-- A-n-CY | DE | | | \|\|\|V1\| |
| CMP r | A-r | r \| A B C D E H L<br>BF B8 B9 BA BB BC BD | | | \|\|\|V1\| |
| CMP M | A-(HL) | BE | | | \|\|\|V1\| |
| CPI n | A-n | FE | | | \|\|\|V1\| |
| INR r | r <-- r+1 | r \| A B C D E H L<br>3C Ø4 ØC 14 1C 24 2C | | | \|\|\|VØ. |
| INR M | (HL) <-- (HL)+1 | 34 | | | \|\|\|VØ. |

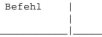
Mikrobefehlsliste - 8ØØØ Operationskodes Blatt 6

| Befehl | Operation | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|
| DCR r | r <-- r-1 | r \| A B C D E H L<br>---\|---<br>3D Ø5 ØD 15 1D 25 2D | | | \|\|\|V1. |
| DCR M | (HL) <-- (HL)-1 | 35 | | | \|\|\|V1. |
| Allgemeine_Arithmetik-_und_Steuerbefehle | | | | | |
| DAA | Dezimalkorrektur nach Operation mit gepackten Zahlen | 27 | | | \|\|\|P.\| |
| CMA | A <-- $\overline{A}$ | 2F | | | ..1.1. |
| CMC | CY <-- $\overline{CY}$ | 3F | | | ..X.Ø\| |
| STC | CY <-- 1 | 37 | | | ..Ø.Ø1 |
| NOP | keine Operation | ØØ | | | ..... |
| HLT | CPU im Haltzustand | 76 | | | ..... |
| DI | IFF <-- Ø | F3 | | | ..... |
| EI | IFF <-- 1 | FB | | | ..... |

Mikrobefehlsliste - 8ØØØ Operationskodes Blatt 7

| Befehl | Operation | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNC V |
|---|---|---|---|---|---|
| 8-bit-Logikbefehle | | | | | |
| ANA r | A <-- A&r | r \| A B C D E H L<br>----\|----------<br>\|A7 AØ A1 A2 A3 A4 A5 | | | \|\|1PØØ |
| ANA M | A <-- A&(HL) | A6 | | | \|\|1PØØ |
| ANI n | A <-- A&n | E6 | n | | \|\|1PØØ |
| ORA r | A <-- Avr | r \| A B C D E H L<br>----\|----------<br>\|B7 BØ B1 B2 B3 B4 B5 | | | \|\|ØPØØ |
| ORA M | A <-- Av(HL) | B6 | | | \|\|ØPØØ |
| ORI n | A <-- Avn | F6 | n | | \|\|ØPØØ |
| XRA r | A <-- AØr | r \| A B C D E H L<br>---\|----------<br>\|AF A8 A9 AA AB AC AD | | | \|\|ØPØØ |
| XRA M | A <-- AØ(HL) | AE | | | \|\|ØPØØ |
| XRI n | A <-- AØn | EE | n | | \|\|ØPØØ |

| Befehl | Operation | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|
| 16-bit-Arithmetik | | | | | |
| DAD d | HL <-- HL+d | d \| BC DE HL SP<br>---\|---------- <br>Ø9 19 29 39 | | | ..X.Ø\| |
| INX d | d <-- d+1 | d \| BC DE HL SP<br>---\|---------- <br>Ø3 13 23 33 | | | ..... |
| DCX d | d <-- d-1 | d \| BC DE HL SP<br>---\|---------- <br>ØB 1B 2B 3B | | | ..... |
| Verschiebebefehle | | | | | |
| RLC |  | Ø7 | | | ..Ø.Ø\| |
| RRC | | ØF | | | ..Ø.Ø\| |
| RAL |  | 17 | | | ..Ø.Ø\| |

| Befehl | Operation | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|
| RAR |  | 1F | | | ..Ø.Ø\| |
| Sprungbefehle | | | | | |
| JMP nn | PC <-- nn | C3 | $n_L$ | $n_H$ | ..... |
| Jcc nn | | cc | | | |
| | Z = Ø | NZ \| C2 | $n_L$ | $n_H$ | ..... |
| | Z = 1 | Z \| CA | $n_L$ | $n_H$ | ..... |
| | C = Ø | NC \| D2 | $n_L$ | $n_H$ | ..... |
| | C = 1 | C \| DA | $n_L$ | $n_H$ | ..... |
| | P = Ø | PO \| E2 | $n_L$ | $n_H$ | ..... |
| | P = 1 | PE \| EA | $n_L$ | $n_H$ | ..... |
| | S = Ø | P \| F2 | $n_L$ | $n_H$ | ..... |
| | S = 1 | M \| FA | $n_L$ | $n_H$ | ..... |
| PCHL | PC <-- HL | E9 | | | ..... |

Mikrobefehlsliste - 8Ø8Ø Operationskodes Blatt 10

| Befehl | Operation | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|
| CALL nn | (SP-1)<-- $PC_H$ | | | | |
| | (SP-2)<-- $PC_L$ | CD | $n_L$ | $n_H$ | |
| | PC <-- nn | | | | |
| | SP <-- SP-2 | | | | |
| Ccc nn | (SP-1)<-- $PC_H$ | | | | |
| | (SP-2)<-- $PC_L$ | | | | |
| | PC <-- nn | | | | |
| | SP <-- SP-2 | | | | |
| | | cc \| BC | | | |
| | Z = Ø | NZ \| C4 | $n_L$ | $n_H$ | |
| | Z = 1 | Z \| CC | $n_L$ | $n_H$ | |
| | C = Ø | NC \| D4 | $n_L$ | $n_H$ | |
| | C = 1 | C \| DC | $n_L$ | $n_H$ | |
| | P = Ø | PO \| E4 | $n_L$ | $n_H$ | |
| | P = 1 | PE \| EC | $n_L$ | $n_H$ | |
| | S = Ø | P \| F4 | $n_L$ | $n_H$ | |
| | S = 1 | M \| FC | $n_L$ | $n_H$ | |

Mikrobefehlsliste - 8Ø8Ø Operationskodes Blatt 11

| Befehl | Operation | 1.Byte | 2.Byte | 3.Byte | Flags SZHPNCV |
|---|---|---|---|---|---|
| RST p | (SP-1)<-- $PC_H$ | p \| Ø 1 2 3 4 5 6 7 | | | |
| | (SP-2)<-- $PC_L$ | \|C7 CF D7 DF E7 EF F7 FF | | | |
| | $PC_H$ <-- Ø | | | | |
| | $PC_L$ <-- p | | | | |
| | SP <-- SP-2 | | | | |
| Ein-und_Ausgabebefehle | | | | | |
| IN n | A <-- n | DB | n | | ..... |
| OUT n | n <-- A | D3 | n | | ..... |

Zeichenerklaerung:

| | |
|---|---|
| X | Flag unbestimmt |
| \| | Flag entsprechend dem Ergebnis der Operation gesetzt/rueckgesetzt |
| Ø | Flag rueckgesetzt |
| 1 | Flag gesetzt |
| P | gerade Paritaet |
| V | Ueberlauf |
| CY | Carry - Flag |
| . | Flag wird nicht veraendert |

$n_L$ niederwertiges Byte der Adresse nn

$n_H$ hoeherwertiges Byte der Adresse nn

$d_L$ ($PC_L$) niederwertiger Teil des Doppelregisters

$d_H$ ($PC_H$) hoeherwertiger Teil des Doppelregisters

## 2. LINK

Um ein in der Assemblersprache geschriebenes Programm unter dem Betriebssystem SCP lauffaehig zu machen, muessen sowohl der Assembler ASM als auch der Programmverbinder LINK benutzt werden.

Der Prozess der Programmentwicklung laeuft in vier Schritten ab:

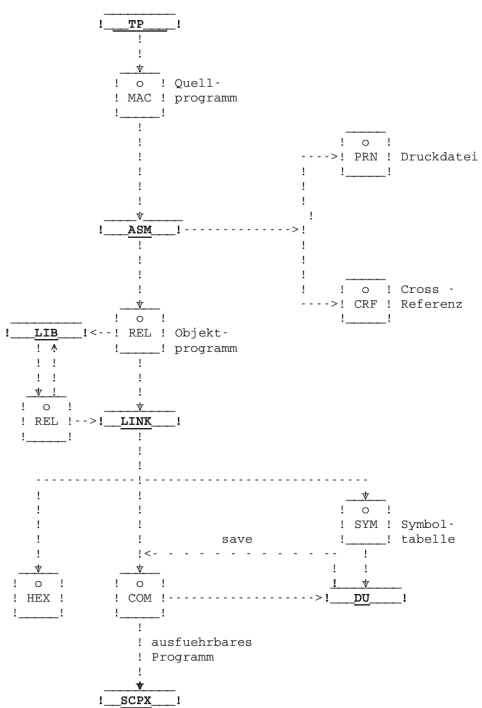
1. Erfassen des in Assemblersprache geschriebenen Programmes (Dateityp MAC) unter Benutzung des Textverarbeitungsprogramms TP.

2. Uebersetzen des Quellprogrammes mit dem Assembler ASM. Das Ergebnis ist ein Objektprogramm (ein Modul) mit dem Dateityp REL in einem dem Maschinencode nahen, aber nicht abarbeitbaren Zwischencode.

3. Laden und Verbinden einzeln uebersetzter Objektprogramme zu einem Gesamtprogramm (Dateityp COM) mit dem Programmverbinder LINK. LINK wandelt die Objektprogramme, die sich auch in einer mit dem Programm LIB gebildeten Bibliothek befinden koennen, zu einem einzigen Programm im Maschinencode um, das unter SCP ausgefuehrt werden kann.

4. Test des lauffaehigen Programmes mit dem Testhilfsprogramm DU.

**Ablauf_der_Mikroprogrammentwicklung**



Die folgende Beschreibung der Arbeitsweise des Programmverbinders LINK verwendet Begriffe, die im Assemblerhandbuch erklaert sind. Die dort benutzte Syntax-Notation wurde beibehalten.

Die vom Assembler erzeugten Programme sind nicht ausfuehrbar. Um ein REL-Programm ausfuehrbar zu machen, muss es mit dem Programmverbinder LINK geladen und gebunden werden.

Laden bedeutet die physische Ablage des Programmes im Speicher und das Umwandeln von Relativadressen in Absolutadressen. Dies ist einer der notwendigen Schritte, um ein verschiebliches Objektprogramm (REL) in ein ausfuehrbares Programm (COM) zu wandeln.

Binden heisst, dass jedes Objektprogramm, das externe Bezuege besitzt (externe Symbole) mit dem entsprechenden Objektprogramm "verbunden" wird, in dem diese Bezuege definiert sind.

LINK kann das uebersetzte und gebundene Programm als ein ausfuehrbares Programm (Dateityp COM) auf Diskette speichern oder eine Datei im Intel-ASCII-HEX-Format (Dateityp HEX) erzeugen.
LINK belegt einen Speicherbereich von 16 K Byte.

## 2.1. Aufruf des Programmverbinders LINK

Durch die Tastatureingabe von

```
LINK          <ET>
```

wird das Programm LINK.COM geladen. LINK zeigt durch einen Stern (*) an, dass es eine Kommando-Eingabe erwartet. Die zu verbindenden REL-Programme muessen im Direktzugriff zur Verfuegung stehen. Ist nur ein Laufwerk verfuegbar und die REL-Programme sind auf mehreren Disketten gespeichert, so muss ein Wechsel der Disketten vor den entsprechenden Lade-Schritten von LINK erfolgen.

## 2.2. LINK-Kommandos

LINK-Kommandos bestehen aus Dateinamen und Schalter. Man kann die LINK-Kommandos nacheinander einzeln eingeben oder alle Kommandos (einschliesslich des Aufrufes des Programmverbinders LINK selbst) in einer Zeile.

Eine Kommando-Zeile hat ein flexibles Format, es koennen wahlweise Angaben fuer das Laden und Verbinden der Programme und fuer andere Operationen eingegeben werden.

Die Grundregel ist dabei, dass die Objektprogramme in der Reihenfolge geladen werden, in der sie angefuehrt sind, beginnend ab der Startadresse Ø1Ø3H. Selbst wenn die Programme in der eingegebenen Reihenfolge geladen werden, braucht man sie nicht in der Folge ihrer Abarbeitung anzugeben, denn LINK erzeugt an der Stelle Ø1ØØH - Ø1Ø2H einen unbedingten Sprung an den logischen Programmanfang.

LINK kann ueber 11 verschiedene Aufgaben ausfuehren, aber in der praktischen Arbeit werden i. a. nicht mehr als drei oder vier gleichzeitig benoetigt.

Nach der Eingabe eines LINK-Kommandos meldet sich LINK mit einem Stern (*) und zeigt an, dass es eine weitere Kommando-Eingabe erwartet.

Beispiel:

```
A>LINK              <Kommandos durch ET beendet>
*/<schalter>
*<dateiname>
*/<schalter>
*/E                 (Verlassen von LINK)
```

Alle Kommando-Zeilen koennen als eine Zeile eingegeben werden:

```
LINK /<schalter>,<dateiname>/<schalter>/E
```

Obwohl die Eingabe eines jeden Kommandos als gesonderte Zeile zeitaufwendig ist, hat sie den Vorteil, dass man zu jedem Zeitpunkt weiss, was LINK gerade durchfuehrt.

### 2.2.1. Dateinamen

Die von LINK verarbeiteten Dateien sind REL-Dateien. Die Eingabe eines Dateinamens bewirkt, dass LINK das entsprechende Objektprogramm laedt. Wenn vorher schon ein Programm geladen wurde, fuehrt LINK das Binden der Objektprogramme aus.

```
A>LINK                      <ET>
*REL1,REL2,REL3             <ET>
```

Im einfachsten Fall benoetigt LINK die Eingabe des Dateinamens der zu verarbeitenden REL-Datei und den Namen des zu bildenden ausfuehrbaren Programms (i.a. eine COM-Datei).

```
A>LINK                      <ET>
*REL1,COMDAT/N/E            <ET>
```

Wird waehrend des LINK-Laufes nur ein Dateiname eingegeben, so wird entweder das COM-Programm nicht gebildet oder LINK antwortet mit der Fehlermeldung: ?NOTHING LOADED.

Beachte:
Wird nur ein Dateiname eingegeben, dem jedoch ein /G-Schalter folgt, so wird zwar das COM-Programm nicht gebildet, aber das Programm wird nach dem Laden und Verbinden ausgefuehrt (siehe Erlaeuterungen in Abschnitt 2.2.2.).
Man kann so viele Dateinamen und Schalter in einer Kommando-Zeile eingeben, wie es die Kapazitaet der Zeile erlaubt.

Wenn ein LINK-Lauf beendet ist, wird i.a. das ausfuehrbare Programm unter dem angegebenen Namen gerettet (Dateiname, gefolgt von einem /N-Schalter, siehe 2.2.2.). LINK vergibt fuer diesem Dateinamen den Dateityp COM.

Ein Dateinamen-Kommando bei LINK besteht im allgemeinen aus der Laufwerkbezeichnung, dem Dateinamen und dem Dateityp.

<laufwerk>:<dateiname>.<typ>

LINK nimmt standardmaessig als <laufwerk> das aktuelle logische Disketten-Laufwerk, als <typ> fuer die Eingabe REL und fuer die Ausgabe COM an.

**Beispiel:**
Sollen mehrere Objektprogramme von verschiedenen Laufwerken zu einem ausfuehrbaren Programm verbunden werden, und die entstehende COM-Datei (COMDAT) auf Laufwerk B: abgelegt werden, so sind z.B folgende Kommandos erforderlich:

```
A>LINK                          <ET>
*REL1,B:REL2,C:REL3,REL4        <ET>
*B:COMDAT/N/E                   <ET>
```

Die Objektprogramme REL1 und REL4 werden von Laufwerk A: gelesen, waehrend REL2 von Laufwerk B: und REL3 von Laufwerk C: geholt werden.

## 2.2.2. Schalter

Die Schalter-Kommandos von LINK fuehren Funktionen ohne Laden und Verbinden aus. Schalter sind Buchstaben, denen ein Schraegstrich (/) vorangestellt ist. Man kann in einer Kommandozeile so viele Schalter wie benoetigt angeben, aber jedem einzelnen Schalter-Buchstaben muss ein Schraegstrich (/) vorangehen.

Wenn zum Beispiel ein Programm namens TEST geladen und gebunden, das entstandene Programm unter dem gleichen Namen auf Diskette gerettet und dann ausgefuehrt werden soll, benoetigt man zwei Dateinamen und zwei Schalter:

```
*TEST,TEST/N/G          <ET>
```

LINK speichert das geladene und gebundene Programm auf Diskette (/N-Schalter), dann wird es abgearbeitet (/G).
Einige Schalter koennen allein eingegeben werden (/E, /G, /R, /P, /D, /U, /M, /O, /H, /X). Vor einigen Schaltern muss ein Dateiname stehen (/N, /S).

Der /Y-Schalter wirkt nur, wenn andere Schalter beim LINK-Lauf eingegeben wurden.

Einige Schalter sind nur vor der Eingabe eines Dateinamens sinnvoll (/P, /D). Andere Schalter-Kommandos werden erst am Ende des LINK-Laufes ausgefuehrt (/N, /Y, /X). Weitere Schalter-Kommandos werden unmittelbar nach ihrer Eingabe ausgefuehrt (/S, /R). Diese "Regeln" sollten bei Eingabe des LINK-Kommandos beachtet werden (siehe detaillierte Beschreibung der "Schalter").

**Beachte:**
**Die LINK-Schalter wirken anders als die mit dem gleichen Buchstaben bezeichneten ASM-Schalter!**

**Uebersicht_LINK-Schalter**

| Funktion | Schalter | Wirkung |
|---|---|---|
| **Ausfuehren** | **/G** | Ausfuehrung des gebundenen Programms, dann zurueck zum Betriebssystem. Falls /N gegeben war, wird die COM-Datei auf Diskette geschrieben. |
| | **/G:<name>** | Beginn der Abarbeitung des gebundenen Programmes bei der Adresse, die dem Wert von <name> (globales Symbol !) entspricht. Falls /N spezifiziert war, wird die COM-Datei auf Diskette geschrieben. |
| **Beenden** | **/E** | Verlassen LINK und zurueck zum Betriebssystem. Falls /N gegeben war, wird zuvor die COM-Datei geschrieben. |

| Funktion | Schalter | Wirkung |
|---|---|---|
| | **/E:<name>** | Setzt die Startadresse des COM-Programmes gleich dem Wert von <name> und geht ins Betriebssystem. Falls /N gegeben war, wird zuvor die COM-Datei geschrieben. |
| **Retten** | **/N** | Vergabe des Namens der zu bildenden COM- bzw. HEX-Datei. Der Dateiname muss vor /N stehen! |
| **Adresse setzen** | **/P** | Setzt die Anfangs-Ladeadresse fuer Programme und Daten. Wenn auch der /D-Schalter benutzt wird, setzt /P nur die Programm-Ladeadresse. |
| | **/D** | Setzt die Anfangsadresse nur fuer Datenbereiche. |
| | **/R** | Setzt LINK auf den Anfangszustand. |
| **Bibliotheks-suche** | **/S** | Durchsucht die Bibliothek, deren Name unmittelbar vor /S angegeben ist. |
| **Listen der Globalen** | **/U** | Listet die undefinierten globalen Symbole. |
| | **/M** | Listet die gesamte Cross-Referenz-Tabelle. |
| **Festlegen Zahlendarstellung** | **/O** | Oktal. |
| | **/H** | Hexadezimal (Standard). |
| **Spezieller Code** | **/Y** | Erzeugt eine spezielle Datei fuer den symbolischen Test mit DU, fordert /N- und /E-Schalter, gibt der speziellen Datei die Typbezeichnung SYM. |
| | **/X** | Erzeugt statt der COM-Datei eine Datei im Intel-ASCII-HEX-Format. |

Die drei am haufigsten benutzten Schalter sind /G (Beenden mit Ausfuehren), /E (Beenden) und /N (Retten Programm).

## Ausfuehren

### /G-Schalter

Der /G-Schalter veranlasst LINK, die geladenen und gebundenen Objektprogramme als Gesamtprogramm auszufuehren. Nach dem Programmlauf erfolgt die Rueckkehr ins Betriebssystem.
Wurde der /N-Schalter angegeben, so wird das Programm vor Ausfuehrung als COM-Datei auf Diskette geschrieben.
Ebenso wird bei gesetztem /Y-Schalter eine Symboldatei erstellt.

Beispiel:

```
A>LINK TEST,TEST/N/G          <ET>
```

bindet TEST.REL, rettet das Ergebnis als Disketten-Datei TEST.COM und startet das Programm TEST.COM.

Bevor die Ausfuehrung beginnt, gibt LINK drei Zahlen und den Text BEGIN EXECUTION aus.
Die erste Zahl ist die Startadresse des Programmes. Die zweite Zahl gibt die Adresse des naechsten verfuegbaren Bytes an, das ist die Endadresse des Programms + 1. Die dritte Zahl ist die Anzahl der 256-Byte-Seiten, die das Programm belegt.

Soll das COM-Programm nicht gerettet werden, gibt man nur den Dateinamen der REL-Datei und den /G-Schalter in der Kommando-Zeile an:

Beispiel:

```
A>LINK TEST/G          <ET>
```

Beachte:
Wird kein COM-Programm auf Diskette erzeugt (/N weggelassen), muss das auszufuehrende Programm fuer eine erneute Abarbeitung mit LINK neu gebunden werden.

### /G:<name>

Der /G:<name>-Schalter besitzt die gleiche Wirkung wie der /G-Schalter, aber mit einer zusaetzlichen Funktion.

<name> ist ein in einem der geladenen und gebundenen Objektprogramme definiertes globales Symbol. LINK nimmt <name> als Startpunkt des Programms an und traegt die entsprechende Adresse in den Sprungbefehl auf 1ØØH - 1Ø2H ein.

Die Bedeutung dieses Schalters liegt darin, dass LINK das Programm an einem vorgegebenen Punkt startet, wenn der Startpunkt in den assemblierten Objektprogrammen nicht eindeutig definiert ist.

Normalerweise ist dies kein Problem, wenn ein Objektprogramm von einer hoeheren Programmiersprache (das LINK standardmaessig als Hauptprogramm ansieht) eingebunden wird, oder man bindet nur durch ASM erzeugte Objektprogramme und genau eines hat eine END <name>-Anweisung zur Definition des Startpunktes.

Aber in den Faellen, wo zwei oder mehr END <name>-Anweisungen vorhanden sind oder kein Objektprogramm eine solche enthaelt, gibt /G:<name> dem Programmverbinder LINK den logischen Programmanfang fuer das Ausfuehren des gebundenen Programmes.

## Beenden

### /E-Schalter

/E zeigt das Ende eines LINK-Laufes an. Das Programm LINK wird verlassen und zum Betriebssystem zurueckgekehrt. Wurde der /N- oder der /Y-Schalter benutzt, so wird zuvor eine entsprechende COM- bzw. SYM-Datei auf Diskette erstellt.

Wenn das Binden beendet ist, gibt LINK drei Zahlen aus: Startadresse, Adresse naechstes verfuegbares Byte, Anzahl der 256-Byte-Seiten.

### /E:<name>

wirkt genau wie der /E-Schalter, definiert aber zusaetzlich durch <name> einen Startpunkt. Im uebrigen gilt das gleiche wie bei /G:<name> (siehe oben).

## Retten

### /N-Schalter

Der /N-Schalter bewirkt die Bekanntgabe des Namens der zu erstellenden Programm-Datei auf Diskette. Dieser Dateiname muss unmittelbar vor /N stehen. Ist kein Dateityp angegeben, wird standardmaessig COM angenommen.

Das eigentliche Retten des gebundenen Programms auf Diskette erfolgt erst mit dem Erkennen der /E- bzw. /G-Schalters.

Soll ein einziges Objektprogramm zu einem ausfuehrbaren Programm gebunden werden, so ist z.B folgende Kommandozeile erforderlich:

```
A>LINK TEST,TEST/N/G        <ET>
```

Der erste Name TEST bezeichnet das zu ladende und zu bindende Objektprogramm (eine REL-Datei), der zweite Name ist der Name der als Ergebnis des LINK-Laufes entstehenden und zu rettenden COM-Datei.
Natuerlich ist es moeglich, Dateinamen in beliebiger Reihenfolge anzugeben, z.B.:

```
A>LINK TEST/N,MOD1,MOD2,PROG/G      <ET>
```

Hier laedt und bindet LINK die Objektprogramme MOD1,MOD2 und PROG, dann wird das entstandene Programm unter dem Namen TEST gerettet und das Programm wird gestartet.

**Beachte:**
**Wird als Name des zu rettenden Programmes ein Name angegeben, der eine schon auf der gleichen Diskette gespeicherte Datei bezeichnet, so wird die vorhandene Datei ueberschrieben und ist somit verloren.**

**/N:P-Schalter**

LINK rettet im Normallfall den Programmbereich und den Datenbereich in die COM-Datei. Sollte es einmal notwendig sein nur den Programmbereich zu retten (um Platz auf der Diskette zu sparen), dann kann dazu der Schalter /N:P benutzt werden.
Soll z.B. nur der Programmbereich unter dem Namen TEST.COM auf Diskette abgespeichert werden, so ist folgende Kommandozeile erforderlich:

```
A>LINK MOD1,TEST/N:P/E          <ET>
```

Die Benutzung der folgenden Schalter gestatten einige zusaetzliche Funktionen, die eine gezielte Beeinflussung der LINK-Ablaeufe ermoeglichen.

**Adresse_setzen**

Jedes in Assemblersprache geschriebene Programm kann sich aus absoluten (ASEG) und relativen, d.h. verschieblichen (COMMON, DSEG, CSEG) Programmteilen zusammensetzen (s. Beschreibung im Assemblerhandbuch).

Laedt LINK ein solches Programm in den Speicher, so geschieht das standardmaessig wie folgt:

Unabhaengig von der phyischen Anordnung der einzelnen Programmteile (ASEG, COMMON, DSEG, CSEG) im Assemblerprogramm erfolgt die Speicherung in der Rangordnung

1. COMMON - Programmteile
2. DSEG - Programmteile
3. CSEG - Programmteile

beginnend bei Adresse 1Ø3H (1ØØH - 1Ø2H enthaelt den unbedingten Sprung an den logischen Programmanfang bzw. den Wert ØØØØØH).

Beispiel:

Besitzt ein ASM-Quellprogramm die Struktur
(dseg1,cseg1,common1,cseg2,dseg2,common2),
so erzeugt LINK ab 1Ø3H ein wie folgt strukturiertes COM-Programm:

(common1,common2,dseg1,dseg2,cseg1,cseg2).

ASEG - Programmteile werden in den entsprechenden vorgegebenen Speicherbereich geladen, dabei ist darauf zu achten, dass sich keine Ueberlagerungen mit anderen Programmteilen ergeben.

Weitere durch LINK geladene Objektprogramme werden lueckenlos an das vorher geladene Objektprogramm angefuegt, wobei jedes Programm in sich in obiger Rangordnung gespeichert wird.

Beispiel:

Aus Programmen P1 und P2 mit der Struktur
P1: (dseg1,cseg1,dseg2)
P2: (cseg2,common1,dseg3,cseg3)
erzeugt LINK durch das Kommando

*P1,P2 <ET>

folgende Struktur:

(dseg1,dseg2,cseg1,common1,dseg3,cseg2,cseg3).

COMMON-Bloecke gleichen Namens werden stets in den gleichen Speicherbereich geladen, den der erste COMMON-Block dieses Namens durch LINK zugewiesen erhaelt. Er legt somit die Laenge des Bereiches fest, und das entsprechende Programm muss zuerst geladen werden.

Die Verwendung der /P- und /D- Schalter ermoeglicht einen von dieser Standardstrukturierung abweichenden Aufbau des von LINK erzeugten COM-Programmes.

Die /P- und /D-Schalter legen absolute Adressen innerhalb des COM-Programmes fest. Dies kann z.B. zum Trennen von Programm- und Datenbereich innerhalb des erzeugten Gesamtprogrammes oder zum Festlegen von fuer den Test guenstigen Anfangsadressen der gebundenen Objektprogramme genutzt werden.

Grundsaetzlich gilt bei Verwendung von /P- und /D-Schalter:

- Die Schalter wirken nur auf nachfolgende Objektprogramme, d.h. bereits geladene Programme werden nicht beeinflusst.

- Auf ASEG-Programmteile ueben diese Schalter keine Wirkung aus.

- Waehrend eines LINK-Laufes koennen mehrere /P- und /D-Schalter angegeben werden.

- Luecken zwischen gebundenen Programmen werden durch LINK nicht initialisiert, sondern koennen Daten oder Programmteile eines vorhergehenden Programmes enthalten. Dies kann unter Umstaenden die Ursache fehlerhafter Ergebnisse bei Abarbeitung des erzeugten Programmes sein.

- Das Format der Schalter ist

  ```
          /P:<adresse>,
  ```

  bzw.

  ```
          /D:<adresse>,
  ```

  Die Adresse muss in der aktuellen Zahlendarstellung angegeben werden. Die Standarddarstellung ist hexadezimal.

**/P-Schalter**

Werden waehrend des LINK-Laufes nur /P-, aber keine /D-Schalter angegeben, so legt ein /P-Schalter die Ladeadresse des nachfolgenden zu bindenden Objektprogrammes fest. Innerhalb des Objektprogrammes erfolgt die oben angefuehrte Umstrukturierung.
Der Standard-Anfangswert fuer den /P-Schalter ist 1Ø3H.

Beispiel (s.o.):

```
     *P1,/P:3ØØ,P2     <ET>

     erzeugt:

     (dseg1,dseg2,cseg1)                ab 1Ø3H
     und  (common1,dseg3,cseg2,cseg3)   ab 3ØØH
```

Damit koennen die Objektprogramme an vorgegebenen Adressen im Gesamtprogramm angeordnet werden.

Werden beide, /P- und /D-Schalter verwendet, so beeinflusst der /P-Schalter nur die CSEG-Teile der nachfolgenden Objektprogramme.

**/D-Schalter**

Der /D-Schalter wirkt nur auf die COMMON- und DSEG-Teile der nachfolgenden zu bindenden Objektprogramme.

Beispiel (s.o.):

Die LINK-Kommando-Zeile

```
*/D:2ØØ,P1,P2            <ET>
```

erzeugt:

```
(cseg1,cseg2,cseg3)                    ab 1Ø3H !

und  (dseg1,dseg2,common1,dseg3)       ab 2ØØH
```

Damit kann eine Trennung von Programm- und Datenteil erfolgen. Es ist auch moeglich, Programm- und Datenteil ab vorgegebenen Adressen anzuordnen, z.B. Programmteil ab 4ØØH und Datenteil ab 2ØØH:

```
*/D:2ØØ,/P:4ØØ,P1,P2    <ET>
```

LINK bindet die geladenen Objektprogramme entsprechend den angegebenen Anfangsadressen zu einem Gesamtprogramm. Nicht erlaubt ist das Mischen der **durch Schalter** getrennten Bereiche, d.h. beispielsweise die Kommando-Folge:

```
*/P:2ØØ,/D:3ØØ,P1,P2    <ET>
*/P:4ØØ,P3              <ET>
```

wuerde ein CSEG ab Ø2ØØH, DSEG ab Ø3ØØH und CSEG ab Ø4ØØH erzeugen. Dies ist nicht moeglich und fuehrt zu einer Fehlermeldung.

Zusaetzliche Bemerkung zu /P- und /D-Schalter:

Wenn das entstehende Programm den verfuegbaren Speicherplatz ueberschreitet, ist es in einigen Faellen durch die gemeinsame Benutzung von /D- und /P-Schalter doch moeglich, die zu ladenden und zu verbindenden Objektprogramme in den Speicher zu laden. Waehrend LINK laedt und bindet, bildet es eine Lade-Tabelle, die aus fuenf Byte fuer jede Eintragung besteht. Durch das Setzen beider Schalter, /D und /P, entfaellt das Bilden dieser Tabelle und somit steht ein zusaetzlicher Arbeitsspeicherbereich zur Verfuegung.

**/R-Schalter**

Der /R-Schalter setzt LINK auf den Anfangszustand wie nach dem Aufruf zurueck. Wird ein /R-Schalter erkannt, werden alle geladenen Objektprogramme ignoriert, d.h. es ist kein Objektprogramm geladen und die /P- und /D-Schalter werden auf den Anfangszustand (Adresse Ø1Ø3H) gestellt. LINK initialisiert sich selbst und meldet sich mit "*" bereit zur neuen Kommandoeingabe.

## Bibliothekssuche

**/S-Schalter**

Der /S-Schalter durchsucht die Datei (REL-Programm(e)), deren Name unmittelbar vor dem Schalter angegeben ist, nach Routinen, Subroutinen, Definitionen fuer globale Symbole usw.

Bei der Eingabe der Kommando-Zeile muss der Dateiname mit dem angehaengten /S-Schalter vom Rest der Zeile durch ein Komma getrennt werden, z.B.:

```
A>LINK TEST/N,ULIB/S,TEST/G        <ET>
```

Der /S-Schalter wird benutzt, um Programme zu suchen, die durch das Programm LIB zu einer Bibliothek zusammengefasst wurden.

## Listen_der_globalen_Symbole

**/U-Schalter**

Der /U-Schalter bewirkt das Listen aller undefinierten globalen Symbole. Er wirkt nur in einer Kommando-Zeile, die weder einen /G-, noch einen /E-Schalter enthaelt.

Beachte, dass LINK automatisch alle undefinierten Globalen listet, ausser wenn die Kommando-Zeile auch einen /S-Schalter enthaelt. In diesem Fall gibt man den /U-Schalter ein und erhaelt so diese Liste.

Das Listen kann durch die Betaetigung der Taste ^S (control S) unterbrochen und mit ^Q (control Q) fortgesetzt werden.
Zusaetzlich zum Listen der undefinierten Globalen werden durch den /U-Schalter der Anfangsladepunkt, das Ende und die Groesse des gemeinsamen Programm- und Datenbereiches ausgegeben. Sind beide Schalter, /P und /D, gesetzt, werden Anfangspunkt, Ende und Groesse beider Bereiche gesondert gelistet.

**/M-Schalter**

Durch den /M-Schalter erfolgt das Listen aller Globalen, der definierten und der undefinierten. Dabei steht neben den definierten Globalen ihr Wert (Adresse), die undefinierten sind durch einen Stern (*) gekennzeichnet.

Auch hier werden wie beim /U-Schalter Anfang, Ende und Groesse von Programm- und Datenbereich zusaetzlich gelistet.

## Festlegen_der_Zahlendarstellung

**/O-Schalter**

Der /O-Schalter bewirkt das Umstellen der aktuellen Zahlendarstellung auf Oktal.

**/H-Schalter**

Der /H-Schalter setzt die aktuelle Zahlendarstellung auf Hexadezimal zurueck. Hexadezimal ist die Standard-Darstellung bei LINK, deshalb braucht man den /H-Schalter nur anzugeben, wenn man vorher /O eingegeben hatte und zur hexadezimalen Darstellung zurueck will.

## Symboldatei

**/Y-Schalter**

Der /Y-Schalter erzeugt eine Symbolnachweisdatei, die von dem Testhilfsprogramm DU fuer den symbolischen Test benutzt werden kann.

**Der /Y-Schalter verlangt den /N- und den /E- oder /G-Schalter z.B.:**

```
A>LINK TEST,TEST/N/Y/E      <ET>
```

Als Dateiname der Symbolnachweisdatei wird stets der Name vor dem /N-Schalter benutzt. Das Dateiattribut ist jedoch SYM. In der Symbolnachweisdatei sind die Namen und Adressen der globalen Symbole gespeichert, die im Gesamtprogramm definiert wurden.

## HEX-Datei

**/X-Schalter**

Wurde der /X-Schalter gesetzt, so wird statt der COM-Datei eine Datei im Intel-ASCII-HEX-Format erzeugt.
Der Dateiname der zu bildenden Datei wird mit dem /N-Schalter festgelegt. Ist beim /N-Schalter kein Dateityp angegeben, dann wird HEX angenommen.

Das Kommando

```
          A>LINK REL1,TEST/N/X/E          <ET>
```

erzeugt eine Datei TEST.HEX

## 2.3. Fehlermeldungen

Bei waehrend des LINK-Laufes auftretenden Fehlern erfolgt eine entsprechende Ausschrift, der meist das Zeichen ? oder % vorangestellt ist.

**?No Start Address**

Es wurde der /G-Schalter benutzt, aber kein Objektprogramm geladen.

**?Loading Error**

Keine von LINK verarbeitbare, korrekt formatierte REL-Datei (Objektprogramm) geladen.

**?Out of Memory**

Verfuegbarer Arbeitsspeicherplatz reicht zum Laden der Objektprogramme nicht aus.

**?Command Error**

Kein gueltiges LINK-Kommando.

**?<dateiname> Not Found**

Die REL-Datei mit dem im Kommando angegebenen Namen <dateiname> existiert nicht auf der Diskette.
Eventuell wurde die Laufwerkbezeichnung vergessen.

**?Start Symbol - <name> - Undefined**

Der im /E:<name> oder /G:<name> angegebene Startpunkt existiert nicht als globales Symbol.

**?Nothing Loaded**

Ein <name>/S oder /E oder /G wurde angegeben, aber kein Objektprogramm (REL-Datei) geladen. Das heisst, es wird versucht, eine Bibliothek zu durchsuchen, LINK zu verlassen oder ein Programm auszufuehren, obwohl kein Objektprogramm geladen wurde z.B.

```
A>TEST/N/E          <ET>
```

ergibt ein "?Nothing Loaded", weil TEST/N ein Programm TEST.COM benennt, aber das Laden eines REL-Programmes fehlt.

**?Can't Save Object File**

Diskettenfehler waehrend des Schreibens des zu rettenden Programmes. Normalerweise bedeutet dies, dass die Diskette gefuellt oder schreibgeschuetzt ist.

**%2nd COMMON larger xxxxxx**

Wenn Objektprogramme geladen werden, die COMMON-Bloecke (siehe Assemblerhandbuch) enthalten, reserviert LINK einen Speicherbereich entsprechend der Laenge des ersten geladenen COMMON-Blockes.

Enthaelt nun ein nachfolgendes Objektprogramm einen COMMON-Block mit einer groesseren Laenge, so antwortet LINK mit dieser Fehlerausschrift. Das bedeutet, dass der erste geladene Block mit der Bezeichnung xxxxxx nicht der groesste Block gleichen Namens war. Man muss entweder die Ladereihenfolge vertauschen oder die COMMON-Bloecke umbenennen.

**%Mult. Def. Global YYYYYY**

Der Global- (PUBLIC) Symbol-Name YYYYYY ist in den geladenen Objektprogrammen mehrfach definiert.

**&Overlaying Program Area**

```
,Start       = xxxx
,Public      =<symbolname> (xxxx)
,External    =<symbolname> (xxxx)
```

Normalerweise erfolgt diese Ausschrift, wenn entweder /D oder /P eine Adresse innerhalb des durch LINK belegten Bereiches definieren. Man sollte dann die Schalter-Adressen ruecksetzen.
Diese Ausschrift erfolgt auch, wenn nach bereits geladenen Objektprogrammen ein Schalter mit einer zu niedrigen Adresse fuer ein noch zu ladendes Objektprogramm angegeben wird, so dass eine Ueberlagerung und somit ein Zerstoeren von Teilen der bereits geladenen Objektprogramme erfolgen wuerde.

**%Overlaying Data Area ,Start = xxxx**
**,Public = <symbolname> (xxxx)**
**,External = <symbolname> (xxxx)**

Die /D- und /P-Schalter waren so gesetzt, dass eine Ueberlagerung der Bereiche erfolgen wuerde. Wenn z.B. bei /D eine hoehere Adresse als bei /P angegeben wird, die aber noch im Programmbereich liegt oder wenn die /D-Adresse kleiner als die /P-Adresse ist, aber der Startpunkt des Programmes noch im Datenbereich liegt, dann erfolgt diese Fehlerausschrift.

**?Intersecting Program Area**

oder

**Intersecting Data Area**

Programm- und Datenbereiche ueberschneiden sich und einer im Ueberschneidungebereich liegenden Adresse kann kein aktueller Wert zugewiesen werden.

**Origin Above Loader Memory, Move Anyway (Y or N)?**

oder

**Origin Below Loader Memory, Move Anyway (Y or N)?**

Diese Ausschrift erfolgt nur bei einem /E- oder /G-Schalter. Steht LINK nicht genuegend Speicherplatz zum Laden eines Objektprogramms zur Verfuegung, aber ein /E oder /G wurde nicht eingegeben, wird man die "?Out of Memory"-Ausschrift erhalten.

Erfolgt ein Speicherueberlauf, weil die zu ladenden Objektprogramme zu gross fuer den verfuegbaren Speicherplatz sind oder ein /D- oder /P-Schalter in den Systembereich zeigt, gibt LINK die "Origin Above Loader Memory"-Ausschrift.

Wird ein /D- oder /P- Schalter unterhalb von 1ØØH gesetzt, erfolgt die "Origin Below Loader Memory"-Ausschrift. Damit wird verhindert, dass das Program in den fuer das Betriebssystem bestimmten Speicherbereich geladen wird.
Wird Y <ET> eingegeben, erfolgt ein Transport des Bereiches und Weiterarbeit. Wird irgend etwas anderes eingegeben, so wird LINK beendet. In jedem Fall erfolgt, wurde der /N-Schalter eingegeben, eine Rettung des Programmes auf Diskette.

## 3. LIB

### 3.1. Aufruf des Bibliothekars

LIB ist ein Dienstprogramm zum Bilden und Verwalten einer Bibliothek aus in Assemblersprache geschriebenen Objektprogrammen. LIB kann auch als Laufzeit-Bibliotheksverwalter fuer hoehere Programmiersprachen eingesetzt werden.

**Anmerkung:**
**Vor der Benutzung von LIB ist die Anfertigung einer Sicherheitskopie der zu behandelnden Bibliothek zu empfehlen, da man auch leicht Bibliotheken zerstoeren kann.**

LIB wird durch das Kommando

```
LIB            <ET>
```

aufgerufen. LIB meldet sich mit einem Stern (*), um seine Bereitschaft zur Kommandoeingabe anzuzeigen. Jedes Kommando in LIB fuegt Objektprogramme zu der sich im Aufbau befindlichen Bibliothek hinzu.

LIB kann Laufzeit-Bibliotheken aus Objektprogrammen bilden. Diese Objektprogramme koennen Unterprogramme fuer Compiler hoeherer Programmiersprachen oder fuer andere Assemblerprogramme sein.

Unter einer Bibliothek soll eine Zusammenfassung von mehreren Objektprogrammen zu einer neuen Objektprogramm-Datei verstanden werden.
Der Vorteil der Bildung einer solchen Bibliothek liegt darin, dass alle fuer die Ausfuehrung eines Programmes benoetigten Routinen mit ihm zu einem ausfuehrbaren Gesamtprogramm (COM-Datei) vom Programmverbinder LINK gebunden werden koennen.
Dazu sind nur der Bibliotheksname, gefolgt von einem /S in der LINK-Kommando-Zeile einzugeben.

Beispiel:

```
A>LINK HPRG,BIB1/S,PROG/N/G      <ET>
```

Dies ist bequemer als die einzelne Eingabe der Objektprogramme, besonders wenn ihre Anzahl gross ist. Bei Verwendung einer Bibliotheksdatei kann man sicher sein, dass alle benoetigten Objektprogramme in die COM-Datei gebunden werden, ausserdem besteht keine Gefahr, dass die Kommandozeile nicht ausreicht. Schliesslich steht die zu einer Bibliothek zusammengefasste Sammlung von Unterprogrammen fuer das Einbinden in beliebige Programme zur Verfuegung.

LIB belegt einen Speicherbereich von etwa 5 K Byte.

### 3.2. Anwendungsfaelle

Die zwei meist benoetigten Anwendungsfaelle von LIB sind **Bilden** einer Bibliothek und **Listen** einer Bibliothek. Die folgenden Beispiele zeigt die Grundkommandos fuer diese beiden Aufgaben.

#### 3.2.1. Bilden einer Bibliothek

Beispiel:

```
A>LIB                              <ET>
*MATLIB=SIN,COS,TAN,COTAN          <ET>
*EXP                               <ET>
*/E                                <ET>
```

In diesem Beispiel wird durch das SCP-Kommando LIB der Bibliothekar LIB aufgerufen, der sich mit einem Stern (*) als bereit zur Kommandoeingabe meldet.

MATLIB ist der Name der Bibliothek, die gebildet werden soll. Auf Diskette wird eine leere Datei MATLIB.LIB erzeugt. SIN, COS, TAN, COTAN sind die Namen der Objektprogramme, die zu MATLIB verkettet werden sollen.

EXP ist der Name eines anderen Objektprogramms das zu MATLIB gehoeren soll. (EXP koennte auch in der vorhergehenden Kommandozeile angegeben werden; dieses Beispiel zeigt, dass die Objektprogrammnamen einzeln oder auch mehrere zusammengefasst eingegeben werden koennen.)

/E bewirkt das Umbenennen von MATLIB.LIB in MATLIB.REL und die Ausgabe auf Diskette, wobei die vorher gebildete leere Datei MATLIB.LIB ueberschrieben wird und das Verlassen von LIB.

#### 3.2.2. Listen einer Bibliothek

Beispiel:

```
A>LIB                              <ET>
*MATLIB.LIB/U                      <ET>
*MATLIB.LIB/L                      <ET>
.
.
.
(Liste der Symbole in MATLIB.LIB)
.
.
.
*^C  (CTRL C)
A>
```

In diesem Beispiel erfolgt ebenfalls der Aufruf des Bibliothekars durch das Kommando LIB.

MATLIB.LIB/U bewirkt, dass MATLIB.LIB nach solchen Cross-Referenzen durchsucht wird, die beim ersten Durchlauf durch die Bibliothek nicht aufgeloest werden koennen ("rueckwaerts" gerichtete Referenzen).

MATLIB.LIB/L listet die Namen und Symboldefinitionen der in MATLIB.LIB enthaltenen Objektprogamme.
(^C) kehrt zu SCP zurueck, ohne eine Bibliothek zu beeinflussen!

### 3.3. LIB-Kommandos

Die Kommandos in LIB bestehen aus einem wahlfreien Zielfeld, einem Quellfeld und einem wahlfreien Schalter-Feld.
Das Format eines LIB-Kommandos ist

```
ziel=quelle/schalter
```

#### 3.3.1. Zielfeld

Dieses Feld wird wahlfrei angegeben. Es erfolgt die Angabe des Namens (auch mit Dateityp moeglich) der zu bildenden Bibliothek:

```
bibliotheksname=
```

LIB erzeugt stets voruebergehend eine Datei mit dem Namen bibliotheksname.LIB ! Wird das Zielfeld weggelassen, so wird standardmaessig die Datei FORLIB.LIB erzeugt.
Mit Erkennen eines /E- oder /R-Schalters erhaelt die Datei den in bibliotheksname angegebenen Dateityp. War kein Dateityp gegeben, so wird automatisch REL angenommen.
Falls also eine Datei mit dem Namen bibliotheksname.LIB oder bibliotheksname.REL bereits auf der Diskette existiert, wird diese ohne Warnung ueberschrieben !

#### 3.3.2. Quellfeld

Fuer dieses Feld sind verschiedene Eintragungen moeglich. Die Quellfeld-Eintragungen muessen REL-Dateien sein und bezeichnen Bibliotheken oder Objektprogramme (Module) die man zu einer bestimmten neuen Bibliotheksdatei zusammen fuegen will. Es gibt zwei Eintragungsmoeglichkeiten:

- nur Dateiname(n)
- eine Kombination aus Dateiname(n) und Modulname(n)

**Folgende Syntaxregeln sind zu beachten:**

- Besteht ein Kommando nur aus Dateiname(n), so werden die Eintragungen durch Komma getrennt, z.B.:

```
*PROG1,PROG2,PROG3        <ET>
```

- Besteht ein Kommando aus Datei- und Modulnamen, muessen die Modulnamen in spitze Klammern (<>) eingeschlossen sein. Die Module folgen unmittelbar dem Namen der Datei (Bibliothek), in der sie enthalten sind. Jede Kombination dateiname <modul> wird durch ein Komma von anderen Eintragungen in der Kommando-Zeile getrennt, z.B.:

```
*PROG1,PROG2<UP1>,PROG3<SUM>,PROG4      <ET>
```

- Werden mehr als ein Modul der gleichen Datei angegeben, muessen die in spitze Klammern (<>) eingeschlossenen Modulnamen voneinander durch ein Komma getrennt werden, z.B.:

```
*PROG1,PROG2<UP1,UP2>,PROG3             <ET>
```

Dateien und Moduln sind typische Unter- oder Hauptprogramme in hoeheren Programmiersprachen (z.B. FORTRAN, BASIC) oder Programme in der Assemblersprache, die ENTRY- bzw. GLOBAL- oder PUBLIC-Anweisungen enthalten (Eintrittspunkte).

LIB erkennt einen Modul an seinem Namen, der entweder durch die TITLE- oder NAME-Pseudooperation der Assemblersprache definiert wurde oder der standardmaessig den ersten sechs Zeichen des Dateinamens der Quelldatei entspricht (siehe Assemblerhandbuch).

Alle Quelldateien muessen REL-Dateien sein, LIB kettet eine Datei oder einen Modul an den anderen. So gibt es keinen Unterschied zwischen dem Beispiel unter Syntaxregel 2 und der Folge:

```
*PROG1              <ET>
*PROG2<UP1>         <ET>
*PROG3<SUM>         <ET>
*PROG4              <ET>
```

Es ist zu empfehlen, die Moduln so zu verketten, dass alle Cross-Referenzen "vorwaerts" gerichtet sind. Das heisst, die Moduln, die externe Bezuege enthalten, sollten physisch vor den Moduln angeordnet werden, die die Eintrittspunkte (die Definitionen) enthalten. Im anderen Fall kann LINK, wenn eine Bibliothek durchsucht werden soll, nicht in einem einzigen Durchlauf alle Cross-Referenzen aufloesen.

Ergaenzungen_zu_den_Quell-Moduln

LIB kann Moduln aus vorhandenen Bibliotheken oder anderen REL-Dateien herausloesen.

Es gibt fuer die Definition einer solchen Modulkette folgende Moeglichkeiten:

- **Ein Modul aus einer Bibliothek:**

  *PROG1<UP1>

- **Einige nicht unmittelbar aufeinanderfolgende Moduln einer Bibliothek:**

  *PROG1<UP1,UP2,UP3>

  Allgemein koennen die Moduln in beliebiger Reihenfolge angegeben werden, aber guenstigerweise in einer solchen, wie sie spaeter fuer eine geeignete Ein-Durchlauf-Suche benoetigt werden.

- **Vom ersten Modul der Bibliothek bis zum angegebenen Modul:**

  Angabe zweier Punkte (..) und den Namen des letzten Moduls der Kette.

  *PROG1<..UP1>

- **Von einem angegebenen Modul bis zum letzten Modul der Bibliothek:**
  Angabe des Namens des Anfangsmoduls der Kette, gefolgt von zwei Punkten (..).

  *PROG1<UP1..>

- **Aufeinanderfolgende Moduln einer Bibliothek:**
  Angabe des Namens des ersten Moduls der Kette, gefolgt von zwei Punkten, gefolgt vom Namen des letzten Moduls der Kette

  *PROG1<UP1..UP4>

- **Relative Adressierung eines Moduls der Bibliothek:**
  Angabe eines Modulnamens, gefolgt von einem Plus (+) und der Anzahl der nachfolgenden bzw. einem Minus (-) und der Anzahl der vorhergehenden in die Kette einzubeziehenden Moduln.

Der Modul, dessen Name dabei angegeben wird, ist **nicht** in der Kette enthalten.
Die Modulanzahl muss eine ganze Zahl im Bereich 1-255 sein.

Beispiele:

| | |
|---|---|
| PROG1<UP1+2> | enthaelt die zwei Moduln, die unmittelbar dem Modul UP1 folgen. |
| PROG1<UP3-4> | enthaelt die vier Moduln, die unmittelbar vor dem Modul UP3 liegen. |

Modulreihen und relative Angaben koennen auch gemischt erfolgen, z.B. definiert

```
PROG1<UP1+1..SUM-1>
```

eine Kette aller Moduln zwischen UP1 und SUM, ausser UP1 und SUM selbst.

- **alle Moduln einer Bibliothek:**
  Angabe nur eines Dateinamens

```
PROG1
```

### 3.3.3. Schalterfeld

Eine Eintragung im Schalterfeld bewirkt zusaetzliche LIB-Funktionen. Eine Schalterfeld-Eintragung besteht aus einem Buchstaben, dem ein Schraegstrich (/) vorangeht.

| Schalter | Wirkung |
|---|---|
| **/E** | Verlaesst LIB und bewirkt die Rueckkehr zum Betriebssystem. Soll keine neue Bibliothek gebildet oder eine vorhandene geaendert werden, muss ^C (CTRL C) anstelle /E zur Beendigung benutzt werden.<br><br>Erfolgt keine Angabe des Dateitypes im Zielfeld bei der Kommandoeingabe, erzeugt LIB als Standard den Dateityp LIB. Durch /E wird dieser **standardmaessig** erzeugte Dateityp in REL umbenannt und die neu gebildete Bibliothek auf Diskette ausgegeben.<br><br>Ein bei der Kommandoeingabe **direkt** angegebener Dateityp im Zielfeld wird dagegen nicht veraendert, auch wenn dieser .LIB ist!<br><br>Es wird nochmals nachdruecklich empfohlen, stets einen Dateinamen im Zielfeld der LIB-Kommandozeile anzugeben! |

| Schalter | Wirkung |
|---|---|
| **/R** | /R hat die gleiche Wirkung wie /E, aber bewirkt keine Rueckkehr zum Betriebssytem.<br>Auch hier ist die gleiche Vorsicht wie beim Gebrauch von /E angebracht!<br><br>/R wird anstelle /E nur dann benutzt, wenn unmittelbar im Anschluss die naechste Bibliothek gebildet werden soll. |
| **/L** | Erzeugt eine Liste der Moduln einer angegebenen Bibliothek und der in ihnen enthaltenen Symboldefinitionen (Cross-Referenzen). |
| **/U** | Es werden die Symbole gelistet, die bei einem einfachen Durchlauf durch die Bibliothek undefiniert bleiben wuerden, d.h., alle "rueckwaerts" (zu einem vorhergehenden Modul) weisenden Referenzen. |
| **/C** | /C setzt LIB zurueck, d.h., die eingegebenen Kommandos werden ignoriert, ohne dass LIB verlassen wird. Die im Aufbau befindliche Bibliothek wird geloescht und der LIB - Lauf beginnt erneut. LIB meldet sich mit einem Stern (*). /C wird benutzt, wenn falsche Moduln spezifiziert oder eine falsche Reihenfolge eingegeben wurden und ein Neueingabe erfolgen soll. |
| **/O** | setzt Anzeigemodus (Zahlendarstellung) auf Oktalbasis. /O wird gemeinsam mit /L angegeben.<br><br>Beachte:<br>Werden mehrere Schalter angegeben, muss **jedem** Schalter ein Schraegstrich vorangehen, z.B.<br><br>MATLIB/L/O |
| **/H** | setzt Anzeigemodus auf Hexadezimal-Basis zurueck (Standard-Zahlendarstellung). |

## 4. Beschreibung des Debuggers

### 4.1. Aufgaben

Dieses Systemprogramm dient der Inbetriebnahme und Testung von Maschinenprogrammen im Echtzeit-Modus. Dazu sind folgende Funktionskomplexe realisiert:

- Einlesen von Daten aus einer CP/M-Datei in den TPA,
- Listen einzelner TPA-Bereiche,
- Modifikation von TPA-Bereichen
- Vergleichs- und Sucheroperationen der TPA-Daten
- Abarbeitung von Programmteilen im Echtzeitprinzip mit Vorgabe von Haltepunkten und Spurpunkten zur Protokollierung des Ablaufs
- Hilfsfunktionen fuer hexadezimale und Adressenrechnung
- Laden und Aktivieren von zusaetzlichen Hilfsfunktionen

Damit sind Echtzeithaltepunkttechnik, beobachtbare Programmausfuehrung, symbolische Parameterangabe bei Assemblierung, Disassemblierung und Programmausfuehrung moeglich. Durch Nachruesten zusaetzlicher Dienstprogramme und Einbindung in den Debugger ist dessen Leistungsfaehigkeit aufruestbar.

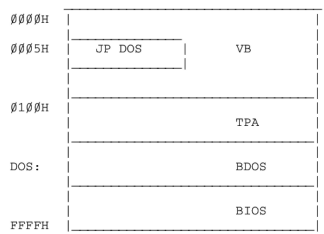
### 4.2. Aufruf des Programmes und Belegung des TPA

Der Debugger wird als transientes Programm durch eines der folgenden CCP-Kommandos

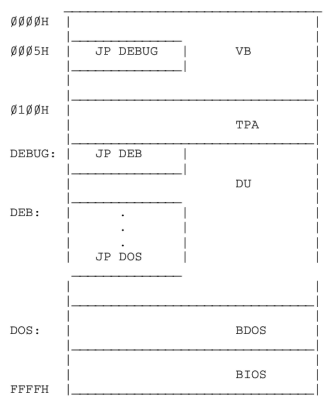
(a) DU
(b) DU_x.y
(c) DU_x.HEX
(d) DU_x.UTL
(e) DU_x.y_u.v
(f) DU_*_u.v

" _ " grafische Darstellung aus einzugebenden Leerzeichen

aufgerufen.
In allen Faellen wird der Debugger als transientes CCP-Programm geladen und die Steuerung an den Debugger uebergeben. Der Debugger ist ein verschiebliches Programm und laedt sich selbst an das obere Ende des TPA. Anschliessend wird die obere Grenze des TPA (im Verstaendigungsbereich Zelle 6/7) unmittelbar unterhalb des Debuggers gesetzt. Damit steht der TPA ab Speicherzelle 1ØØH als Testbereich zur Verfuegung.
Im CCP-Status hat der Speicher folgende Aufteilung:




Das Aufrufformat (a) laedt und aktiviert den Debugger. Das Speicherbild hat dann folgende Struktur:




Die BDOS-Eintrittsadresse (ØØØ6/7H) wird geaendert und der TPA verkleinert.

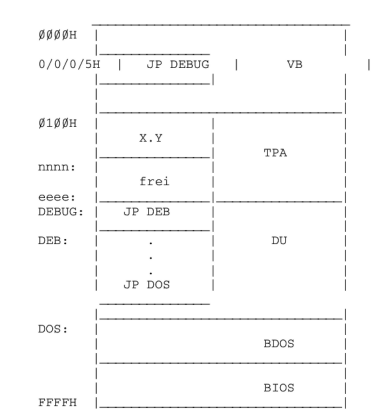
Das Aufrufsformat (b) bewirkt das Laden des Debuggers, das Laden einer Datei x.y in den TPA sowie die Aktivierung des Debuggers. Wenn die Datei x.y nicht auffindbar ist, wird "?" ausgegeben und das Laden der Datei unterdrueckt. Nach erfolgreichem Laden der Datei wird die Meldung

```
NEXT   PC     END
nnnn   pppp   eeee
```

auf der Konsole ausgegeben. Dabei sind nnnn, pppp und eeee hexadezimale Werte, die die Speicheradressen

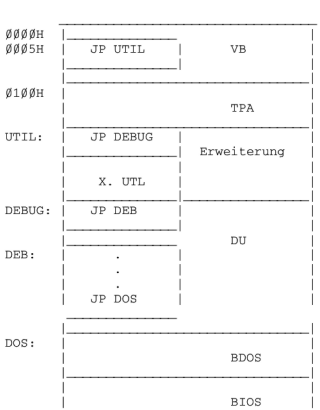
nnnn - naechste freie Adresse nach dem geladenen Programm
pppp - Initialwert des Test-Programmzaehlers
eeee - Endadresse des verfuegbaren TPA (DEBUG - 1)

darstellen. Die Daten des TPA aus der Datei x.y koennen dann mit dem Debugger bearbeitet werden.
Der Speicher ist in der Form




strukturiert.

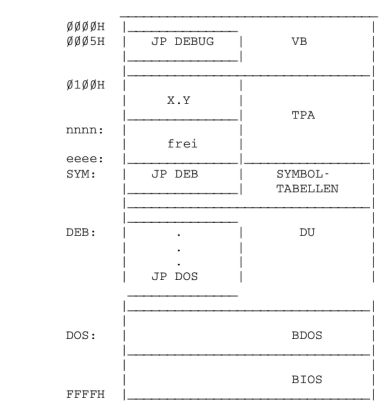
Das Aufrufformat (c) unterscheidet sich von (b) dadurch, das die Dateitypkennzeichnung "HEX" so interpretiert wird, dass die Datei x.HEX im Intel-HEX-Format aufgebaut ist. Dieses wird beruecksichtigt, dass der Debugger zum Laden von x.HEX in den TPA auf einen anderen Lader zurueckgreift. Das HEX-Format kann z.B. durch den Assembler ASM erzeugt werden. Innerhalb der Datei x.HEX koennen auch Informationen ueber den Eintrittspunkt in das Testprogramm enthalten sein. Dann wird beim Laden die PC-Zelle auf diesen Wert gesetzt.
Beim Aufrufformat (d) hat die automatisch nachzuladende Datei die Typvereinbarung "UTL". Damit wird gekennnzeichnet, dass diese Datei eine Erweiterung des Debuggers darstellt und ebenfalls gegenueber dem Aufrufformat (b) einer Sonderbehandlung unterliegt.
Nach Laden und Verschiebung des Debuggers im Speicher wird die Erweiterung nachgeladen und an das obere Ende des TPA geladen und der TPA weiter eingeschraenkt. Die Steuerung wird dann an den Debugger uebergeben. Die Erweiterungen koennen ueber spezielle Unterprogrammaufrufe auf Debuggerkommandoebene aktiviert werden.
Der Speicher ist dann in der Form




strukturiert.

Das Aufrufformat (e) spezifiziert zwei SCP-Dateien. Der erste Dateiname wird als Testprogramm entsprechend Format (b) interpretiert und entsprechend in den TPA geladen. Die zweite Datei u.v wird als Datei interpretiert, die eine Symboltabelle enthaelt. Diese Symboltabelle kann durch den Binder LINK erzeugt werden, enthaelt eine tabellarische Zuordnung von Bezeichnern zu hexadezimalen Werten und hat ein spezielles Datenformat. (Ein Zuordnungspaar besteht aus genau 4 hexadezimalen Zahlen, die vom Symbol durch die Belegung 2ØH getrennt sind. Das zugehoerige Symbol besteht aus bis zu 16 ASCII-Zeichen, die durch eine oder mehrere Tabulator-Zeichen oder durch eine Wagenrueck-lauf - Zeilenschaltung - Sequenz abgeschlossen werden.) Diese Symboltabelle wird unmittelbar unter den Debugger geladen und anschliessend der TPA reduziert. Ein erfolgreiches Laden der Symboltabelle wird durch die Konsolenmeldung "SYMBOLS" gekennzeichnet. Kann diese Datei nicht gefunden werden, wird die Meldung "?" ausgegeben und der Ladevorgang der Datei unterdrueckt.

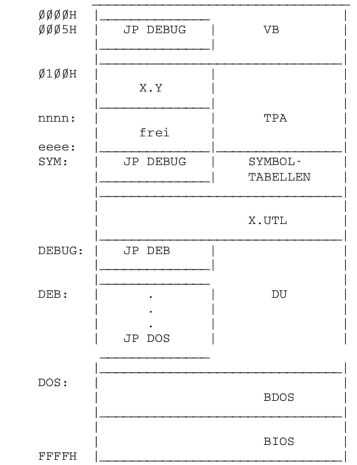
Der Speicher ist dann in der Form




aufgeteilt.

Der Debugger arbeitet auch mit den vorhandenen Symboltabellen. Die Parameter in den Debuggerkommandos koennen dann auch symbolisch angegeben werden und werden automatisch durch die zugeordneten hexadezimalen Werte substituiert. Desweiteren werden beim Listen von hexadezimalen Werten duch den Debugger auch die entsprechenden Symbole mit protokolliert, wenn sie in der Tabelle vorhanden sind.
Beim Aufrufformat (f) wird kein Testprogramm nachgeladen, sondern nur die Symboltabelle geladen.

Das Nachladen von Dateien geschieht durch Angabe der entsprechenden Parameter innerhalb des CCP-Kommandos. Es koennen aber auch mit dem Debugger-Kommando "Read" durch Angabe der gleichen Parameterfolgen entsprechende Dateien in beliebiger Reihenfolge nachgeladen werden. Dabei werden die verschieblichen Programme (x.UTL, u.v) jeweils am oberen Ende des aktuellen TPA abgelegt und der aktuelle TPA entsprechend reduziert. Um die einzelnen Komponenten auch aktivieren zu koennen, muessen sie in einer bestimmten Reihenfolge im Speicher aufgebaut werden.




Die so angeordneten Symboltabellen werden automatisch verkettet.

## 4.3. Aufbau der Debugger-Kommandozeilen

Nach Aufruf des CCP-Kommandos (siehe 4.1.) wird das Debugger-programm in den TPA geladen, anschliessend an das obere Ende des TPA verschoben und der TPA reduziert. Dann wird der Debugger aktiviert. Dieser laedt gegebenenfalls zuerst die Dateien nach, die laut CCP-Aufruf (siehe 4.1.) angefordert werden. Dann wird der Debugger-Kommando-Status erreicht. Auf der Konsole erscheint das Zeichen "#" als Aufforderung an den Bediener, ein Debugger-Kommando einzugeben. Die Kommandoeingabe besteht in einer Serie von Kommandozeilen, die jeweils einzeln sofort nach Abschluss der Zeile abgearbeitet werden.
Grossbuchstaben haben die gleiche Bedeutung wie entsprechende Kleinbuchstaben.
Diese Kommandos umfassen

- Anzeige und Modifikation des Speichers
- Anzeige und Modifikation der virtuellen Test-CPU-Register
- Steuerung der Echtzeitabarbeitung und der Haltepunkt-operationen
- Dateien in den TPA nachladen
- Anlegen von Standard-Dateisteuerbloecken und Parameter-zeilen im Verstaendigungsbereich
- Hilfsfunktionen.

Die Kommandozeile hat eine Kapazitaet von 64 Zeichen. Die Eingabe der Kommandozeile erfolgt mit der BDOS-Funktion #1Ø (siehe dort). Bei der Edition sind folgende Steuerzeichen gueltig:

ctrl-C:
ctrl-E:
ctrl-P:
ctrl-R:

Das Format des Kommandos hat die folgende Form:

```
[Praefix]<Kennzeichen>[Erweiterung][<PAR1>[,<PAR2>[,<PAR3>]]]
```

Das optionale Praefix wird durch das Minuszeichen angegeben und wird als Attribut des Kommandos interpretiert. Das Kennzeichen wird mit einem Buchstaben angegeben und legt die Kommandogruppe fest. Moegliche Modifikation der Kommandogruppe wird durch die optionale Kennzeichenerweiterung als Buchstabe "W" vorgenommen. Diesem Kommando koennen in der Zeile noch bis zu 3 Parameter folgen, die durch Kommata getrennt werden (Statt Kommata koennen auch Leerzeichen benutzt werden). Diese Parameter stellen u.a. hexadezimale Werte im Bereich von Ø - 64k dar.
Die Eingabe der Zeile wird durch die Abschlusstaste abge-schlossen. Sofort darauf wird die Zeile syntaktisch ueber-prueft. Bei positiver Pruefung wird die Kommandozeile interpre-tativ vom Debugger abgearbeitet, anderenfalls wird die Fehler-meldung "?" ausgegeben und die Bearbeitung des Kommandos unter-drueckt. In beiden Faellen wird anschliessend eine neue Komman-dozeile angefordert.

Die Rueckkehr in den CCP-Kommandostatus erfolgt durch Ausloesung eines Systemwarmstarts durch Eingabe von ctrl-C als 1. Zeichen in der Kommandozeile.

## 4.4. Angabe der Parameter

In jeder Kommandozeile koennen auch Parameter angegeben werden. Jeder Parameter stellt einen hexadezimalen Wert dar, der als 8-bit- oder als 16-bit-Wert oder als 16-bit-Adresse interpretiert wird. Ein wichtiger Vorteil dieses Debuggers ist die Moeglichkeit der Zuweisung von numerischen Operandenwerten durch Angabe von symbolischen Ausdruecken. Diese Symbolzuordnung entnimmt der Debugger einer Symboldatei, die in den Arbeitsspeicher geladen werden kann und durch den Binder von Assemblerprogrammen erstellt wird. Die Parameter werden durch Verknuepfung von Operanden gebildet.

### 4.4.1. Operanden

Die Operanden stellen die Grundbestandteile der Parameter dar. Die Standardzahlenbasis des Debuggers fuer Operanden ist die hexadezimale Basis. Die Operandenbasis kann durch vorgelagerte Sonderzeichen umgeschaltet werden.

#### 4.4.1.1. Hexadezimale Zahlen

Der Debugger akzeptiert hexadezimale Ziffern im Bereich von Ø... 9, A,..., F. Aus der Folge dieser Ziffern werden die letzten 4 Ziffern des Operanden ausgewertet und als 16-bit-Wert zugeordnet.

Beispiel fuer Operanden:

| Eingabefolge | Hex-Wert | Dezimalwert |
|---|---|---|
| 1 | ØØØ1 | 1 |
| 2Ø5 | Ø2Ø5 | 517 |
| fffd | FFFD | 65533 |
| 1Ø12d | Ø12D | 3Ø1 |

#### 4.4.1.2. Dezimale Zahlen

Die Standardzahlenbasis des Debuggers ist 16. Die Operandenbasis kann in die dezimale Zahlenbasis durch Vorlagerung des Sonderzeichens "#" umgewandelt werden. Diesem folgen dann innerhalb des Operanden nur Ziffern des Bereiches Ø... 9. Die Ziffern der Folge werden von links her als Hexadezimalzahl konvertiert, deren letzte 16 Bit dem Operandenwert zugeordnet werden.

Beispiel:

| Eingabefolge | Hex - Wert |
|---|---|
| #5 | 5 |
| #263 | Ø1Ø7 |
| #6553Ø | FFFA |
| #65546 | ØØØA |
| #16553Ø | 869A |

#### 4.4.1.3. Zeichenkettenwerte

Die Operatorwerte koennen auch als Zeichenkette dargestellt werden, die durch abbildbare ASCII-Zeichen, in Apostrophe eingeschlossen, angegeben werden. Die ASCII-Werte der letzten beiden Zeichen der Kette werden als 16-bit-Wert dem Operandenwert zugewiesen. Apostrophe koennen nicht nur als Begrenzer, sondern auch als abbildbare ASCII-Zeichen angegeben werden, wenn sie doppelt als Zeichen auftreten.

Beispiel:

| Eingabefolge | Hex-Wert |
|---|---|
| 'A' | ØØ41 |
| 'Ab' | 4162 |
| 'ABCD' | 4344 |
| '_A' | 2Ø41 |
| 'A_' | 412Ø |
| '''' | ØØ27 |
| '''''' | 2727 |
| '''A' | 2741 |
| 'A''' | 4127 |

#### 4.4.1.4. Symbolsche Bezuege

Wenn eine Symboltabelle vom Debugger geladen wurde, kann der Operand auch symbolisch angegeben werden. Der Debugger durchsucht die Symboltabelle nach dem angegebenen Symbol. Der dem ersten gefundenen aequivalenten Symbol zugeordnete Hexwert wird dann herangezogen. Wird das Symbol nicht gefunden, erfolgt eine Meldung fuer fehlerhaftes Debugger-Kommando.
Es sind die 3 Formen der symbolischen Bezugnahme

(a) .<Symbol>
(b) @<Symbol>
(c) =<Symbol>

vorgesehen. Symbole sind alpha-numerische Begriffe mit bis zu 16 Zeichen. Entsprechend der 3 Vorschaltzeichen werden 3 verschiedene Symbolreferenzen unterschieden. Form (a) stellt die Adressreferenz dar. Als Operandenwert wird der dem Symbol zugeordnete Hexwert zugewiesen. Form (b) erzeugt einen 16-bit-Wert, einen "Wortwert", der in den beiden Bytes des Speichers enthalten ist, der durch .<Symbol> symbolisch dargestellt wird. Form (c) erzeugt im Gegensatz zu Form (b) nur den Bytewert anstelle

des Wortwertes. Das hoeherwertige Byte des Operandenwertes ist dann ØØH.

Beispiel:

```
                Symboltabelle:      Ø2ØØ     SYMB1
                                    Ø2Ø2     SYMB2

              Speicherbelegung:     Ø2ØØ  :  12
                                    Ø2Ø1  :  34
                                    Ø2Ø2  :  56
                                    Ø2Ø3  :  78

                  Symbolbezug     Hexwert
                     .SYMB1         2ØØ
                     .SYMB2         2Ø2
                    @SYMB1         3412
                    @SYMB2         7856
                    =SYMB1         ØØ12
                    =SYMB2         ØØ56
```

#### 4.4.1.5. Bedingte Symbolreferenzen

Es besteht die Moeglichkeit, dass mehrere Symboltabellen vom Debugger geladen werden. Diese Tabellen werden entsprechend der Ladefolge miteinander verkettet. Dadurch koennen in der Symboltabelle Symbole mehrfach auftreten. Bei der einfachen Symbolreferenz wird immer die erste gefundene Referenz herangezogen. Um auch auf die anderen mehrfachen Symbolreferenzen zugreifen zu koennen, erlaubt der Debugger auch bedingte Symbolbezuege in der Form

```
<Symbol1>/<Symbol2>/..../<Symboln>
```

Dann durchsucht der Debugger die Symboltabelle vom ersten bis zum letzten Symbol in der Art, dass zuerst das <Symbol1> in der Tabelle gesucht wird. Wird dieses Symbol gefunden, wird ab dieser Stelle das <Symbol2> gesucht usf. bis das letzte Symbol (<Symbol n>) gefunden ist. Der diesem so aufgefundenen Symbol zugeordnete Wert bildet dann die hexadezimale Referenz. Wird bei diesem Suchalgorithmus das Ende der Symboltabelle erreicht, wird Fehlermeldung fuer Debuggerkommando auf der Konsole angezeigt ("?").

Beispiel:

```
                Symboltabelle:      Ø2ØØ   S1
                                    Ø3ØØ   S2
                                    Ø4ØØ   S1
                                    3FCC   S3
                                    1259   S1
                                    Ø2Ø2   S1

              Speicherbelegung:     Ø2ØØ : 12
                                    Ø2Ø1 : 34
                                    Ø2Ø2 : 56
                                    Ø2Ø3 : 78
```

| Symbolbezug | Hexwert |
|---|---|
| .S1 | Ø2ØØ |
| @S1 | 3412 |
| .S1/S1 | Ø4ØØ |
| .S3/S1/S1 | Ø2Ø2 |
| .S2/S1/S1 | 1259 |
| =S3/S1/S1 | ØØ56 |
| @S2/S1/S1/S1 | 7856 |

#### 4.4.1.6. CPU-Register

Der Debugger verwaltet auch eine virtuelle Test-CPU, deren eingestellter Zustand beim Beginn des Echtzeittest der Z80-CPU des Systems zugeordnet wird. Nach Abschluss des Echtzeittest und Uebergabe der Steuerung an den Debugger wird auch der CPU-Zustand an den Debugger uebergeben und stellt dann die aktuelle virtuelle CPU dar. Als Operanden des Debuggers sind die Z80-CPU-Word-Register (16 bit)

| | |
|---|---|
| B - Doppelregister | BC |
| D - Doppelregister | DE |
| H - Doppelregister | HL |
| B'- Doppelregister | B'C' |
| D'- Doppelregister | D'E' |
| H'- Doppelregister | H'L' |
| S - Stackregister | SP |
| P - Programmzaehler | PC |
| X - Indexregister | IX |
| Y - Indexregister | IY |

zugelassen. Durch Voranstellen des Zeichens "*" vor einen der aufgefuehrten Registerkennbuchstaben wird der aktuelle Inhalt des entsprechenden virtuellen CPU-Registers dem Operanden als Wert zugewiesen.

Beispiel:

| Belegung der CPU: | BC=1457 |
|---|---|
| | SP=Ø347 |
| | PC=Ø319 |

| Operand | Hexwert |
|---|---|
| *B | 1457 |
| *S | Ø347 |
| *P | Ø319 |

#### 4.4.1.7. Der Ausdrucksakkumulator als Operand

Ausserdem besteht noch die Moeglichkeit, den aktuellen Parameterakkumulator (siehe 4.4.2.), d.h. den Wert des letzten gueltigen Parameters innerhalb einer Debugger-Dialog-Operation, als Operand aufzurufen. Dieser Operand wird durch das Sonderzeichen "!" angegeben.

Beispiel: Parameterakku : 3F47

| Operand | Hexwert |
|---|---|
| !-2 | 3F45 |

#### 4.4.1.8. Stapeloperanden

Ausser den direkten Operanden existieren auch indirekt adressierbare Operanden. Dabei wird auf den aktuellen Stapelzeiger SP der virtuellen CPU zurueckgegriffen. Es kann in beliebige Tiefe des Kellers ab dieser Stelle zurueckgegriffen werden, ohne den Kellerzeiger zu veraendern. Der Adresszeiger im Keller wird durch das Zeichen "^" (entspricht 5EH) aktiviert. Die Anzahl der ^-Zeichen gibt die Tiefe des Zugriffs im Stapel an. Das dadurch adressierte Datenwort wird als Operandenwert zugewiesen.

Beispiel:

Speicherbelegung:

```
4ØØ : 12
4Ø1 : 34
4Ø2 : 56
4Ø3 : 78
4Ø4 : 9A
4Ø5 : BC
4Ø6 : DE
4Ø7 : FØ
```

Wert des Stapelzeigers: SP : 4ØØ

| Operand | Hexwert |
|---|---|
| ^ | 3412 |
| ^^ | 7856 |
| ^^^ | BC9A |
| ^^^^ | FØDE |
| u.s.w. | |

### 4.4.2. Parameterwerte

Die Operandenwerte (siehe 4.4.1.) koennen durch Hexadezimaladdition und -subtraktion zu Ausdruecken verknuepft werden, die Parameter im Debugger-Kommando darstellen. Nach der Verknuepfung werden die niedrigsten 16-Bit als Parameterwert zugewiesen. Ein gueltiger Parameterwert wird nach seiner Ermittlung durch den Debugger in ein Parameterregister, den Akku eingetragen. Dieser Wert kann bei der Angabe des naechsten Parameters als Operand herangezogen werden.
Es sind beliebig lange additive und subtraktive Verkettungen der Operanden moeglich. Entscheidend ist die Voreinstellung des Parameter-Akkus fuer diese Verknuepfung:

- Beginnt der Parameter mit einem Operand, wird dem Akku dieser Operandenwert zugewiesen.
- Beginnt der Parameter mit einem Pluszeichen, wird zuvor der Parameterakku dem Verknuepfungsakku zugewiesen.
- Beginnt der Parameter mit einem Minuszeichen, wird zuvor der Verknuepfungsakku geloescht.

Beispiel:

```
        Symboltabelle:      Ø413     SY2
                            Ø317     SY1
         CPU-Register:       S       619
                             H       2ØØ
    Speicherbelegung:         619 : 14
                              61A : 75
                              61B : 8A
                              61C : 49
                              61D : 5C
                              61E : AB
                              317 : Ø4
                              318 : 17

     Parameternotation     Parameterwert
                  Ø815          Ø815
             1475+3A           14AF
            6934-19A           679A
       6E1A-#248+.SY1          7Ø39
              +4A9             74E2
               -25A            FDA6
          +123-.SY1            FBB2
          -14C-#417            FD13
                +Ø             FD13
            !+214A             1E5D
             +159C             33F9
            !-33FØ             ØØØ9
            .SY1+!             Ø32Ø
             +@SY1             1A24
              !-*S             14ØB
               +^              891F
              +^^              D2A9
    !-^^^+=SY2/SY1             2751
          +*H-^^               DFC7

              ^^+3             498D
              *S+3             Ø61C
```

## 4.5. Uebersicht ueber Syntax der Debugger-Kommandos

numerische Operanden:

```
<hexziffer> ::= Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

<hexzahl> ::= <hexziffer>
              <hexzahl><hexziffer>

<dezziffer> ::= Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9

<dezzahl> ::= <dezziffer>
              <dezzahl><dezziffer>

<ascii-char> ::= beliebiges abbildbares ASCII-Zeichen

<ascii-ZK> ::= <ascii-char>
               <ascii-ZK><ascii-char>

<num-op> ::= <hexzahl>
             #<dezzahl>
             '<ascii-ZK>'
```

symbolische Operanden:

```
<einf-symb> ::= <ascii-char>
                <einf-symb><ascii-char>

<bed-symb> ::= <einf-symb>/<einf-symb>
               <bed-symb><einf-symb>

<symbol> ::= <einf-symb>
             <bed-symb>

<symb-op> ::= .<symbol>
              @<symbol>
              =<symbol>
```

CPU - Operanden:

```
<cpu-word-reg> ::= B, D, H, B', D', H', X, Y, S, P

<reg-op> ::= *<cpu-word-reg>

<stapel-op> ::= ^
                <stapel-op>^
```

Akku-Operand:

```
<akku-op> ::= !
```

alle Operanden:

```
<operand> ::= <num-op>
              <symb-op>
              <reg-op>
              <stapel-op>
              <akku-op>
```

Parameter:

```
<oz> ::= +
         -

<par> ::= <operand>
          <oz><operand>
          <par><oz><operand>
```

Debugger-Kommando:

```
<CCP-parameter> ::= <ascii-zk>

<par-feld> ::= NIL
               <par>
               <par-feld>,<par>
               <CCP-parameter>

<Kdo-prefix> ::= -

<fkt-erw> ::= W

<fkt-kz> ::= A, C, D, E, F, G, H, I, K, L, M, P, R, S, T, U, X

<Komm> ::= <fkt-kz><par-feld>
           <fkt-kz><fkt-erw><par-feld>
           <Kdo-prefix><fkt-kz><fkt-erw><par-feld>
```

## 4.6. Beschreibung der Kommandos

### 4.6.1. Uebersicht ueber die Funktionen

Die Debuggerkommandos werden ueber die Konsole eingegeben. Die einzelnen Funktionen werden duch das Funktionskennzeichen ausgewaehlt. Die Debuggerfunktionen sind in mehrere Gruppen einteilbar

(a) Dateiarbeit:
    I - Eingabe von CCP-Kommandoparametern, Belegen von Dateisteuerbloecken und Zeichenpuffer im SCPX-Verstaendigungsbereich
    R - Einlesen Dateien in den Arbeitsspeicher des Debuggers

(b) Vergleich, Suche und Modifikation von Daten im TPA-Bereich
    D - Anzeige eines TPA-Bereiches
    S - bytweise Modifikation eines TPA-Bereiches
    F - Belegen eines TPA-Bereiches mit gleicher Belegung
    M - Transfer eines TPA-Datenblockes in einen anderen TPA-Bereich
    E - Vergleich der Belegung zweier TPA-Bereiche
    K - Suche einer Bytefolge in einem TPA-Bereich

(c) Assemblierung und Reassemblierung
    A - Befehlseingabe in mnemonischer Form ueber Konsole in TPA-Bereich
    L - Anzeige eines Programmbereiches in mnemonischer Form

(d) Echtzeittest von Programmen
    P - Setzen, Loeschen und Anzeige der Haltepunkte
    X - Anzeige und Modifikation der Test-CPU
    G - Uebergang in den Echtzeitbetrieb
    T - Ausfuehrung von Befehlsschritten mit Spurprotokollierung
    U - Ausfuehrung von Befehlsschritten ohne Spurprotokollierung

(e) Aufruf von Systemunterprogrammen
    C - Aufruf von zusaetzlichen Systemunterprogrammen

(f) Hilfsfunktionen
    H - Anzeige der Symboltabelle, hexadezimale Addition und Subtraktion

### 4.6.2. Dateiarbeit

#### 4.6.2.1. Uebergabe von CCP-Parametern (I-Kommando)

Wenn ein Programm getestet werden soll, das unter SCPX-Steuerung laeuft, ist es sinnvoll, auch die Parameteruebergabe des CCP-Kommandos an das aufgerufene transiente Programm zu simulieren. Die Uebergabe der CCP-Parameter erfolgt ueber den Verstaendigungsbereich. Die Form des I-Kommandos ist

Icccc.....ccc,

wobei die c eine Folge von ASCII-Zeichen repraesentiert. Diese Folge wuerde normalerweise beim Aufruf des zu testenden Programmes in der CCP-Kommandozeile als Parameterfolge uebergeben. Bei der Uebergabe werden im Verstaendigungsbereich zwei Standard-Dateisteuerbloecke ab Speicheradresse ØØ5CH (DFCB1) bzw. ØØ6CH (DFCB2) angelegt und ausserdem die Parameterzeichenkette im BDOS-Pufferformat (siehe dort) ab Speicheradresse ØØ8ØH (DBUFF) abgelegt.

Beispiele:
(a) IA.B
```
    DFCB1: ØØ 41 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 42 2Ø 2Ø ØØ ØØ ØØ ØØ
    DFCB2: ØØ 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø ØØ ØØ ØØ ØØ
    DBUFF: Ø3 41 2E 42
```

(b) IA:Bcd.EfG_h:IKlM.nO
```
    DFCB1: Ø1 42 43 44 2Ø 2Ø 2Ø 2Ø 45 46 47 ØØ ØØ ØØ ØØ
    DFCB2: Ø8 49 4B 4C 4D 2Ø 2Ø 2Ø 2Ø 4E 4F 2Ø ØØ ØØ ØØ ØØ
    DBUFF: 13 41 3A 42 63 64 2E 45 66 47 2Ø 68 3A 49 4B
           6C 4D 2E 6E 4F ØØ
```

(c) IA*.B?C_D:
```
    DFCB1: ØØ 41 41 3F 3F 3F 3F 3F 42 3F 43 ØØ ØØ ØØ ØØ
    DFCB2: Ø4 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø ØØ ØØ ØØ ØØ
    DBUFF: ØA 41 41 2A 2E 42 3F 43 2Ø 44 3A ØØ
```

(d) IPROG.COM

(e) IPROG.HEX

(f) ITRACE.UTL

(g) IPROG.COM_PROG.SYM

(h) I*_PROG.SYM

#### 4.6.2.2. Einlesen von Dateien (R-Kommando)

Das R-Kommando wird in Verbindung mit dem I-Kommando zum Einlesen von Dateien als Programmsegmente, Symboltabellen und Dienstfunktionen in den transienten Bereich. Als Formen des Kommandos sind

(a) R
(b) Rv

zugelassen. Bei Ausfuehrung des Kommandos wird auf die Belegung der Standard-Dateisteuerbloecke DFCB1 und DFCB2 zurueckgegriffen, die durch das I-Kommando eingestellt werden koennen. Beim Aufruf des Debuggers im CCP-Status (siehe 4.2.) wird die Parameteruebergabe des CCP-Parameter auch ueber die Bloecke DFCB1 und DFCB2 vorgenommen und dann am Anfang der Debuggeraktivierung das R-Programm automatisch abgearbeitet.

Wenn der DFCB2 belegt ist, wird zuerst diese Datei als Symboltabelle interpretiert, direkt unterhalb des Debuggers in den Arbeitsspeicher geladen und der TPA um diesen Symboltabellenbereich reduziert wird. Vorausgesetzt wird, dass diese Datei das Datenformat einer Symboltabellendatei hat. Wenn die Datei erfolgreich geladen ist, wird auf der Konsole die Meldung "Symbols" ausgegeben. Ist die Datei nicht auffindbar oder hat die Datei nicht das erforderliche Datenformat, wird der Ladevorgang unterdrueckt. Beim Laden der Symboltabelle werden die Daten des gesamten TPA nicht veraendert, die Tabelle direkt auf den vorgesehenen Speicherbereich geladen. Durch sequentielles

Laden von Symbol-Dateien werden diese Tabellen zu einer gesamten Tabelle verkettet.

Nach Pruefung und Bearbeitung des DFCB2 wird der DFCB1 und damit die dort spezifizierte Datei bearbeitet. Hat die Datei den Typ .HEX oder .UTL unterliegt sie einer Spezialbehandlung. Anderenfalls wird diese Datei als Datendatei in den TPA geladen. Bei Kommandoform (a) wird diese Datei ab Speicheradresse 1ØØH in den TPA geladen, bei (b) kann noch die Verschiebung v als Parameter angegeben werden, um den die Daten verschoben in den TPA geladen werden. Erscheint auf der Konsole die Meldung "?" ist diese Datei nicht geladen worden. Bei erfolgtem Laden wird anschliessend die Meldung

```
NEXT      PC      END
nnnn      pppp    eeee
```

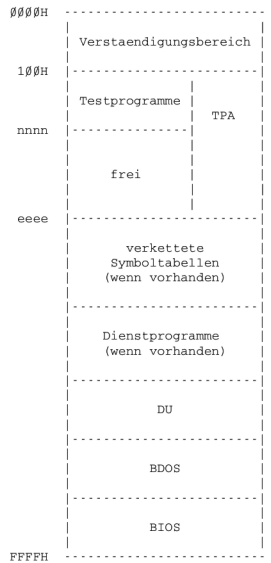
auf der Konsole ausgegeben . Diese 3 Zeiger stellen hexadezimale 4-stellige Adressen dar:

- pppp - aktueller Programmzaehlerstand der Test-CPU
- eeee - Ende des TPA
- nnnn - erste freie Adresse des TPA, die durch R-Kommando noch nicht belegt wurde

Ist die Datei im DFCB1 vom Typ .HEX, wird vorausgesetzt, dass die Daten in Intel-Hex-Format in der Datei stehen. (Diese Datei kann z.B. durch den Binder LINK mit dem Schalter /X erzeugt werden.) Die Kommandoformate (a) und (b) koennen Verwendung finden. Der Zeiger pppp kann beim Laden mit von der Datei uebergeben und eingestellt werden.

Ist die Datei im DFCB1 vom Typ .UTL, handelt es sich um eine Erweiterung der Debugger-Leistung durch ein Dienstprogramm. Dieses Dienstprogramm muss einen speziellen Aufbau haben. Der Ladevorgang beginnt mit dem Einlegen der Datei in den TPA ab der Speicheradresse 1ØØH. Eine Verschiebung v in Kommandoformat (b) wird ignoriert, d.h. (b) wird wie (a) behandelt.
Mit Einlesen der .UTL-Datei wird der TPA ueberschrieben. Anschliessend wird die Steuerung an die Speicheradresse 1ØØH uebergeben und das Programm abgearbeitet. Das Programm ist so aufzubauen, dass es sich selbst an das obere Ende des TPA verschiebt und dann den aktuellen TPA um diesen Betrag reduziert und gegebenenfalls selbststaendig eine Symboltabelle im Speicher aufbaut, die dem Nutzer spaeter den symbolischen Einsprung in die entsprechenden UTL-Funktionen ermoeglicht. Der Ladevorgang wird denn mit einem RET-Befehl im UTL-Programm beendet, der damit die Steuerung an den Debugger zurueckgibt.

Die Reihenfolge der Ladevorgaenge ist vom Nutzer zu organisieren, dass der folgende Aufbau im Arbeitsspeicher vorliegt:




Durch die Anwendung des R-Kommandos werden die Inhalte von DFCB1 und DFCB2 veraendert. Eine wiederholte Anwendung des R-Kommandos setzt jeweils eine Voreinstellung der Bereiche durch das I-Kommando voraus.

### 4.6.3. Datenmanipulation im Arbeitsspeicher

Mit dem Debugger koennen die Daten im gesamten Adressraum des Arbeitsspeicher manipuliert werden. Die Systembereiche mit dem Betriebssystem, dem Verstaendigungsbereich als auch der Debugger selbst sind gegen Veraenderungen nicht geschuetzt. Die Kontrolle ueber diese Bereiche obliegen in diesem Fall dem Nutzer. Ausserdem besteht auch die Moeglichkeit der Edition von Daten in den TPA.

#### 4.6.3.1. Anzeigen eines Speicherbereiches (D-Kommando)

Das D-Kommando dient der Anzeige der Belegung von zusammenhaengenden Bereichen des Arbeitsspeichers in Datenbyte- oder -wortformat. Auf der Konsole. Die Anzeige erfolgt in Hexbelegung als auch in Darstellung der zugeordneten ASCII-Zeichen. Als Kommandos sind

(a) Da
(b) Da,e
(c) D
(d) D,e
(e) DWa
(f) DWa,e
(g) DW
(h) DW,e

Die Formate (a) - (d) bewirken eine Anzeige in Byteform. Die Formate (e) - (h) sind die analogen fuer Anzeige in Wortform, gekennzeichent durch die Funktionserweiterung "W".

Die Anzeige auf der Konsole erfolgt zeilenweise. Eine derartige Doppelzeile hat die Form

```
            aaaa: bb bb ..................... bb
                  c  c                       c
```

im Byteformat;
Das analoge Wortformat kommt in der Form

```
                  aaaa: wwww wwww .... wwww
                        c c  c c ....  c c
```

zur Anzeige.
Die 4 Hexziffern aaaa stellen die Basisadresse der Zeile, d.h. die Speicheradresse des ersten Bytes bzw. Wortes der Zeile dar. Darauf folgen in aufsteigender Reihenfolge der Speicheradressen bis zu 16 Byte in der Form bb bzw. bis zu 8 Datenworte der Form wwww in Hexdarstellung. Zu diesen Hexdarstellungen werden direkt darunter die ASCII-Zeichen c fuer jedes Byte zugeordnet. Wenn die Werte keinem abbildbaren ASCII-Zeichen entsprechen, wird ein Punkt ausgegeben. Die Anzeigezeilen fuer Byteformat werden vom Debugger so aufgeteilt, dass moeglichst die Basisadresse der Zeile Null modulo 16 realisiert wird.

Die Befehlsformate (a) und (e) bewirken eine Darstellung von 6 Anzeige-Zeilen ab der Speicher-Anfansadresse a; die Befehlsformate (b) und (f) veranlassen die Anzeige des Bereiches von Speicheradresse a bis einschliesslich Speicheradresse e. In einer Registerzelle wird bei jeder Anzeige die Adresse nach der letzten angezeigten Belegung abgelegt. Die Formate (c), (d), (g), (h) greifen auf diesen Wert zurueck. Auf diesen Registerinhalt wird als Anfangsadresse fuer den neuen Anzeigebereich Bezug genommen, (c) und (g) fuehren zur Ausgabe einer Anzeigeseite (d.h. 6 Anzeigezeilen), (d) und (h) begrenzen den Bereich durch die Adresse e.

Beispiel:

Wenn ab der Speicheradresse eine Speicherbelegung existiert, die dem Low-Teil der Adressen gleich ist, ergibt das Kommando

```
D3Ø7,325
```

```
Ø3Ø7: Ø7 Ø8 Ø9 ØA ØB ØC ØD ØE ØF
       .  .  .  .  .  .  .  .  .
Ø31Ø: 1Ø 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
       .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
Ø32Ø: 2Ø 21 22 23 24 25
         !  "  #  $  %
```

als Anzeige
und ein folgendes Kommando

```
DW,+13
```

bewirkt

```
Ø326: 2726 2928 2B2A 2D2C 2F2E 313Ø 3332 3534
       ' &  ) (  + *  - ,  / .  1 Ø  3 2  5 4
Ø336: 3736 3938
       7 6  9 8
```

### 4.6.3.2. Fuellung des Speichers (F-Kommando)

Mit diesem Kommando koennen zusammenhaengende Speicherbereiche mit einem konstanten Bytewert belegt werden. Damit koennen Initial- oder Hintergrundbelegungen des Speichers erzeugt werden. Die Kommandoform

```
Fa,e,b
```

erfordert die Angabe von 3 Parametern und veranlasst das Fuellen des Speicherbereiches ab Adresse a bis einschliesslich Adresse e mit der Bytebelegung b. Vom Parameter b wird der Low-Teil des Wertes als Bytebelegung genommen.

### 4.6.3.3. Modifikation von Speicherstellen (S-Kommando)

Diese Funktion erlaubt Daten in den Arbeitsspeicher einzugeben. Als Kommandoformate koennen

(a) Sa
(b) SWa

angegeben werden. Mit Format (a) koennen ab der Speicherstelle a die Bytebelegungen geaendert werden. Nach Aufruf des Kommandos zeigt der Debugger die entsprechende Speicheradresse und die zugehoerige Bytebelegung an. Nun kann ein Parameter eingegeben werden. Der Low-Teil des Wertes wird als Bytebelegung eingetragen. Die Paramertereingabe wird mit der Abschlusstaste abgeschlossen. In der naechsten Konsolzeile erscheint die naechste Speicheradresse und deren Bytebelegung in Hexdarstellung mit Anforderung zur Eingabe eines neuen Parameters. Wird die Eingabe unterdrueckt und nur die Abschlusstaste betaetigt, so bleibt der Wert erhalten und es wird zur naechsten Speicheradresse uebergegangen. Eine Eingabe, die syntaktisch falsch ist, bzw. das Abschlusszeichen .(Punkt) schliessen das S-Kommando ohne Eintragung der letzten, zum Abschluss fuehrenden Eingabe ab. Neben Parametern koennen auch ASCII-Zeichenketten, von einem (") eingeleitet, statt des Parameters eingegeben werden. Dann werden die entsprechenden Werte fortlaufend in den Speicher eingetragen und in den urspruenglichen Modus an der darauffolgenden Speicheradresse wiedereingetreten.

Die Formatform (b) erwartet Modifikation von Datenworten analog (a). Es werden immer ein Datenwort mit entsprechender Adresse angezeigt und der gesamte eingegebene Parameterwert als neues Datenwort uebernommen. ASCII-Ketten werden byteweise entsprechend ihrer Laenge abgearbeitet. Nach Abschluss wird in den Datenwortmodus zurueckgegangen.

Beispiele:

```
Kommando: S2ØØ
        Speicher     Inhalt     Parameter     num. Wert
          Ø2ØØ         C3         123456          56
          Ø2Ø1         9A         #128+14         94
          Ø2Ø2         47         "ABcDe          41
                                                  42
                                                  63
                                                  44
                                                  65
          Ø2Ø6         E5             1           Ø1
          Ø2Ø7         18             .           18
                                                    -->Abschluss
```

```
Kommando:SW3ØØ
        Speicher      Inhalt      Parameter      num. Wert
          Ø3ØØ         457A          1437           37
                                                    14
          Ø3Ø2         5918             4           Ø4
                                                    ØØ
          Ø3Ø4         679B         "ABCDE          41
                                                    42
                                                    43
                                                    44
                                                    45
          Ø3Ø9         AC39        #255+123456      55
                                                    35
          Ø3ØB         475A             .           5A
                                                    47
                                                      -->Abschluss
```

#### 4.6.3.4. Blocktransfer im Arbeitsspeicher (M-Kommando)

Mit diesem Kommando koennen zusammenhaengende Datenbloecke im Speicher ab einer bestimmten Speicheradresse noch einmal als Kopie abgelegt werden. Mit dem Format

M a, e, z

wird ein Datenblock ab der Speicheradresse a bis einschliesslich Speicheradresse e in einen Speicherbereich kopiert, der durch die Anfangsadresse z bestimmt wird.
Der Transport geschieht byteweise in aufsteigender Reihenfolge der Adressen und kann u.a. zu Komplikationen fuehren, wenn die beiden Bereiche nicht durchschnittsfremd und die Zieladresse groesser als die Anfangsadresse, also a<z<e, ist.

Beispiel:

M 4Ø3, 429, 4Ø4 mit Inhalt von 4Ø3=49

entspricht der Funktion

F 4Ø3, 43Ø, 49

#### 4.6.3.5. Vergleich von zwei Datenbloecken (E-Kommando)

Mit diesem Kommando koennen Datenbloecke im Speicher miteinander verglichen werden. Mit dem Format

E a,e,z

wird der Inhalt des Speicherbereiches, der bei Speicheradresse a beginnt und mit der Speicheradresse e endet, mit dem gleichgrossen Speicherbereich verglichen, der mit der Speicheradresse z beginnt.

Auf der Konsole wird ein Protokoll ausgegeben, das die entsprechenden zugeordneten Speicheradressen angibt, bei denen keine Uebereinstimmung der Belegung vorliegt.
Das Protokollformat

aaaa bb cccc dd

gibt in jeder Zeile eine Stelle der Bereiche an, deren Belegungen nicht aequivalent sind. Die Hexzahlen stellen die Adresse aaaa des 1. Bereiches mit der Belegung bb und die entsprechende Adresse cccc des 2. Bereiches und deren Belegung dd dar.

### 4.6.3.6. Suche einer Kette (K-Kommando)

Mit dieser Funktion werden Byteketten in einem zusammenhaengenden Speicherbereich gesucht. Die Anfangsadressen der aufgefundenen aequivalenten Ketten werden protokolliert. Das Kommando

Ka,e,v

sucht in einem Speicherbereich von Adresse a bis e nach Byteketten, die innerhalb dieses Bereiches beginnen. Mit dem Parameter v wird die Speicheradresse der Vergleichskette angegeben. Die Kette aus N Byte hat folgenden Aufbau:

| Speicher: | v+Ø | v+1 | v+2 | ..... | v+N |
|---|---|---|---|---|---|
| Inhalt: | N | B1 | B2 | | BN . |

Beispiel:

```
Speicherbelegung: 3ØØ : 41
                  3Ø1 : 41
                    2 : 42
                    3 : 42
                    4 : 42
                    5 : 43
                    6 : 42
                    7 : 42
                    8 : 42

           Kette: 6ØØ : Ø2
                  6Ø1 : 42
                  6Ø2 : 42

         Befehl: K3ØØ,3Ø6,6ØØ

      Protokoll: Ø3Ø2 Ø3Ø3 Ø3Ø6
```

### 4.6.4. Assembler und Disassembler

#### 4.6.4.1. Assembler (A-Kommando)

Dieses Kommando gestattet, Z80-Maschinencode in den Arbeitsspeicher in mnemonischer Form ueber Konsole einzugeben. Die Operanden werden als Parameter eingegeben. Als Kommandos sind

(a) Aa
(b) A
(c) -A

zugelassen. Format (a) beginnt die Assemblierung ab Adresse a. In jeder Assemblierzeile wird die Befehlsadresse ausgegeben. Dann kann ueber Konsole ein Z80-Befehl in numerischer Form eingegeben werden. Als Trennzeichen zwischen Operationscode und Operandenfeld sind Leerschritte und Tabulatoren zugelassen. Die Befehlsoperanden stellen syntaktisch Debugger-Parameter dar. Die Eingabe des Befehls schliesst mit der Abschlusstaste ab. Nach Eingabe unterzieht der Debugger die Zeile einer syntaktischen Pruefung. Werden falsche Parameterformen verwendet, so wird nach Fehlermeldung "?" das Debuggerkommando verlassen. Bei falscher mnemonischer Darstellung wird der Befehl zurueckgewiesen mit "?" und die Assemblerzeile wird wiederholt.
Bei positiver Syntaxpruefung wird die Zeile assembliert, die Codierung in den Arbeitsspeicher eingetragen und die Adressen fuer naechsten Befehlsbeginn verwaltet. Der Assemblermodus wird verlassen, wenn ein Leerbefehl oder ein .(Punkt) als Befehl eingegeben werden.

Mit der Assemblierung wird eine Zelle, die den aktuellen Adresszeiger beinhaltet verwaltet. die gleiche Zelle wird auch noch von den Debugger-Kommandos U, T und L in analoger Weise verwaltet. Das Format (b) greift auf diese Zelle zu und beginnt die Assemblierung ab der Adresse, die dort eingetragen ist. Bei Verlassen des A-Kommandos kann mit Format (b) ohne Luecke fortgesetzt werden.

Der Assembler-/Disassemblerteil des Debuggers ist ein separater Modul, der am Anfang des Debuggers steht und etwa 1,5kByte umfasst. Mit dem Format (c) wird dieser Teil des Debuggers gestrichen und der TPA bis zum Anfang des verbleibenden Debuggers erweitert. Damit sind alle Daten in den Bereichen, um die der TPA erweitert wurde, nicht mehr im Zugriff (speziell Utilities und Symboltabellen). Damit sind die Befehle A und L nicht mehr ausfuehrbar und alle mnemonischen Darstellungen (z.B. beim X-Kommando) nicht mehr moeglich. Ein Wiederaufruf dieser gestrichenen Teile aus dem Debugger heraus ist nicht moeglich.

#### 4.6.4.2. Disassembler (L-Kommando)

Dieses Kommando disassembliert Maschinencode aus dem Arbeitsspeicher in mnemonische Darstellung. Die Befehle werden befehlsweise als Konsolenzeilen abgebildet. Als Formate sind

```
(a) La
(b) La,e
(c) L
(d) -La
(e) -La,e
(f) -L
```

zugelassen. Format (a) listet das ab Speicheradresse a disassemblierte Mikroprogramm, bis ein Konsolenbild gefuellt ist. Format (b) gibt ausser der Anfangsadresse auch die Endeadresse e an, die einschliesslich zum Bereich gehoert. Diese Adresse ist bytegenau anzugeben. Format (c) setzt die Disassemblierung ueber ein Konsolenbild fort. Das Protokoll hat die Form:

```
SSS....S:
AAAA  BBBB  CCCC..C  .DD....D

mit: AAAA    - Mikrobefehlsadresse
     SSS..S  - Symbolreferenz zu AAAA, wenn sie existiert
     BBBB    - Operationscode
     CCCC    - Operandenfeld
     DDD..D  - Symbolreferenz des Operanden, wenn sie existiert
               bzw. bei Speicheroperationen den Bytewert der
               Adresse vor Ausfuehrung der Operation.
```

Die Formate (d) - (f) sind analog zu (a) - (c), nur werden die symbolischen Bezuege in der Liste unterdrueckt.

### 4.6.5. Hilfsfunktionen (H-Kommando)

Diese Funktion ermoeglicht Berechnungen, Konvertierungen und Anzeige von Parameterwerten. Als Kommandoformate sind

```
(a) Ha,b
(b) Ha
(c) H
```

zugelassen.
Format (a) berechnet die hexadezimale Summe und Differenz der beiden Parameter und bewirkt die Anzeige

```
SSSS  DDDD
```

mit

```
SSSS   Summe a+b in Hexdarstellung
DDDD   Differenz a-b in Hexdarstellung
```

Ein Ueberlauf wird nicht beruecksichtigt.

Format (b) wird zur Konvertierung von Parameterwerten benutzt. Der Parameter a wird in der Form

```
HHHH  SS.....  #DDDDD  'C'
```

protokolliert. Dabei ist

```
HHHH   Hexwert
SS.... Symbolische Darstellung
DDDDD  Dezimalwert
C      zugeordnetes ASCII-Zeichen
```

Als symbolische Darstellung wird die erste Referenz in der Symboltabelle herangezogen. Existiert keine Referenz, so wird die symbolische Darstellung unterdrueckt. Das ASCII-Zeichen wird auch nur protokolliert, wenn es darstellbar ist.

Format (c) listet die gesamte Symboltabelle in der Reihenfolge, wie sie geladen wurde und bei Zugriffen durchsucht wird.

### 4.6.6. Aufruf von Systemunterprogrammen (C-Kommando)

Mit diesem Kommando werden Systemunterprogramme aufgerufen. Bei Abarbeitung dieser Programme wird der Zustand der Test-CPU nicht beeinflusst. Diese Unterprogramme treten bei Anwendung von Dienstprogrammen auf. Es werden besonders Unterprogramme unterstuetzt, denen ueber die CPU-Register B, C, D, E Parameter uebergeben werden. Die Steuerung an dem Debugger wird mit RET zurueckgegeben, Parameter koennen aber ueber die CPU-Register nicht zurueckgegeben werden.

Als Kommandoformate sind

```
(a) Ca
(b) Ca,b
(c) Ca,b,d
```

zugelassen.
Format (a) arbeitet ein Unterprogramm ab, das ab der Arbeitsspeicheradresse a beginnt. Vor Eintritt in das Programm werden die CPU-Register BC:=DE:=ØØØØ gesetzt.
Format (b) setzt vor Eintritt das Registerpaar BC:=b und DE:=ØØØØ,
Format (c) setzt vor Eintritt das Registerpaar BC:=6 und DE:=c

## 4.6.7. Echtzeittest von Programmen

### 4.6.7.1. Anzeige und Modifikation der CPU-Register (X-Kommando)

Dieses Kommando wird zur Pruefung und Voreinstellung des CPU-Zustandes vor Eintritt in den Echtzeittest herangezogen.
Die Formate

```
(a) X
(b) Xf   mit f = C, Z, M, E, I, C', Z', M', E', I'
(c) Xr   mit r = A, B, D, H, S, P, A', B', D', H', X, Y
```

sind zugelassen.
Das Format (a) zeigt den CPU-Zustand zweizeilig auf der Konsole in der Form

```
CZMEI A=AA B=BBBB D=DDDD H=HHHH S=SSSS P=PPPP
CZMEI A'aa B'bbbb D'dddd H'hhhh X=xxxx Y=yyyy  cc....c
```

an. In der oberen Zeile stehen die Inhalte der CPU-Register der Haupt-Register-Seite und in der unteren Zeile stehen die Erweiterungen fuer U880 (2. Registerseite und Indexregister) sowie der naechste Befehl.

Es beideuten:

```
AA    - Inhalt Register A
aa    - Inhalt Register A'
BBBB  - Inhalt Registerpaar BC
bbbb  - Inhalt Registerpaar B'C'
DDDD  - Inhalt Registerpaar DE
dddd  - Inhalt Registerpaar D'E'
HHHH  - Inhalt Registerpaar HL
hhhh  - Inhalt Registerpaar H'L'
xxxx  - Inhalt Indexregister IX
yyyy  - Inhalt Indexregister IY
SSSS  - Inhalt Stapelzeiger SP
PPPP  - Inhalt Programmzaehler PC
cccc  - Befehl auf der Adresse PPPP
C     - Carryflag C ist gesetzt/geloescht   (C/-)
Z     - Zeroflag Z                          (Z/-)
M     - Minusflag M                         (M/-)
E     - Paritaetsflag E                     (E/-)
I     - Halbcarryflag I                     (I/-)
c     - Carryflag C'                        (C/-)
z     - Zeroflag Z'                         (Z/-)
m     - Minusflag M'                        (M/-)
e     - Paritaetsflag E'                    (E/-)
i     - Halbcarryflag I'                    (I/-)
```

Der anstehende Befehl wird in mnemonischer Darstellung abgebildet. Wenn der Disassembler nicht im Debugger ist (durch -A abgeschaltet), wird der Befehl im Hex-Code dargestellt.
Das Format (b) dient der Modifikation der CPU-Flags. Nach Eingabe des Befehls erscheint in der naechsten Konsolenzeile der vorliegende Zustand des Flags, wird das entsprechende f angezeigt, ist das Flag gesetzt; erscheint "-" ist das Flag

geloescht. Durch Eingabe eines Parameters, der den Wert ØØØØ hat, wird das Flag zurueckgesetzt. Der Wert ØØØ1 setzt das Flag. Alle anderen Werte brechen die Ausfuehrung des Kommandos ab. Wird kein Parameter eingegeben, bleibt das Flag unveraendert.

Das Format (c) dient der Modifikation der CPU-Register. Nach Eingabe des Befehls erscheint in der naechsten Konsolenzeile der Wert des Registers bzw. Registerpaares. Die Eingabe eines Parameters setzt das Register auf dessen Wert. Wird kein Parameter eingegeben, bleibt der alte Wert erhalten.

#### 4.6.7.2. Modifikation von Passpunkten (P-Kommando)

Ein Passpunkt wird im Programm gesetzt, um den Durchlauf durch diesen Punkt beim Echtzeittest zu protokollieren. Aehnlich den temporaeren Unterbrechungspunkten erfolgt ein Ruecksprung in den Debugger-Mode. Der Durchlauf wird vom Debugger registriert und zwischen Beendigung und Fortsetzung des Echtzeittest entschieden, abhaengig davon, ob der Durchlaufzaehler, der bei jedem Passpunktdurchlauf dekrementiert wird, den Wert Null erreicht hat oder nicht. Im Gegensatz zu den temporaeren Unterbrechungspunkten, bleiben die Passpunkte beim Abbruch des Echtzeit-Mode bestehen.
Mit diesem Kommando koennen Passpunkte gesetzt, geloescht und angezeigt werden. Als Formate sind

```
(a) Pa
(b) Pa,z          ; z= 1, ..., 255
(c) P
(d) -Pa
(e) -P            zugelassen.
```

Mit Format (a) wird ein Passpunkt auf die Adresse a gesetzt. Der zugehoerige Durchlaufzaehler wird auf den Wert Eins gesetzt. Wird der Durchlaufpunkt im Test erreicht, wird der Echtzeitmode nach Abarbeitung des zur Adresse a gehoerigen Befehls abgebrochen, der Zaehler behaelt seinen Wert, der Passpunkt bleibt erhalten.

Mit Format (b) wird ein Passpunkt auf die Adresse a gesetzt und der zugehoerige Durchlaufzaehler auf den Wert z voreingestellt. Der Wert z=Ø hat die gleiche Wirkung wie Format (d).

Es lassen sich bis zu 8 Passpunkte setzen.

Mit dem Format (c) wird die Liste der bestehenden Passpunkte auf der Konsole angezeigt. Jede Konsolzeile

```
ZZ  AAAA .SS..S
```

mit

ZZ - Stand des Durchlaufzaehlers
AAAA - Adresse
SS..S - moegliche Symbolreferenz zu AAAA

wird einem Passpunkt zugeordnet.

Mit dem Kommando (d) wird ein Passpunkt auf der Adresse a geloescht. Mit dem Kommando (e) werden alle existierenden Passpunkte geloescht.

Wenn ein Passpunkt beim Testlauf durchlaufen wird, wird der Echtzeitmodus verlassen und die Information

ZZ PASS aaaa .SS-S

mit

ZZ - Durchlaufzaehlerstand
aaaa - Adresse des Passpunktes
SS-S - Symbolreferenz zu aaaa

gefolgt von einer Protokollierung des CPU-Zustandes (wie bei X-Kommando) auf der Konsole ausgegeben.
Anschliessend wird der Durchlaufzaehler dekrementiert. Hat der Zaehler damit den Wert Null erreicht, wird er auf den Wert Eins gesetzt und ein temporaerer Unterbrechungspunkt hinter den naechsten Befehl gesetzt. Anderenfalls unterbleibt diese Aktion. In beiden Faellen wird dann der Testlauf fortgesetzt.
Im ersten Fall wird nach Abarbeitung des naechsten Befehles dann der Testlauf automatisch bei Erreichen des Unterbrechungspunktes beendet.

### 4.6.7.3. Uebergang in den Echtzeitbetrieb (G-Kommando)

Mit diesem Kommando gibt der Debugger die Steuerung an das zu testende Programm ab, das dann unter Echtzeitbedingungen ablaeuft und getestet werden kann. Erscheint im Programm ein RST 7, so wird der Testlauf beendet und die Steuerung an den Debugger zurueckgegeben. Dieser RST 7 kann im Debugger-Mode an jeder beliebigen Stelle des Testprogrammes gesetzt werden; es muss nur beim Ablauf des Programmes als solcher Befehl interpretiert werden. Vor Eintritt in den Echtzeit-Mode subsituiert der Debugger automatisch an den Adressen, die durch Passpunkte oder temporaere Unterbrechungspunkte gekennzeichnet sind, den Inhalt der Speicherstelle durch die Belegung FFh (RST 7). Mit Abarbeitung des RST 7 wird die Steuerung an den Debugger zurueckgegeben. Zunaechst werden vom Debugger die urspruenglichen Belegungen an den Haltepunkten wieder eingetragen und damit das Speicherbild des Programmes wieder hergestellt.

Bei Eintritt in den Test wird der Satz der virtuellen CPU-Register (siehe Kommando X) in die CPU des Geraetes geladen und mit der Befehlsabarbeitung begonnen. Bei Uebergang aus dem Echtzeit- in den Debugger-Mode wird der erreichte Zustand der CPU als virtuelle CPU uebernommen.

Als Formate sind

(a) G
(b) Ga
(c) Ga,b
(d) Ga,b,c
(e) G,b
(f) G,b,c
(g) -G....

zugelassen.

Format (a) bewirkt einen Uebergang in den Echtzeitmode an der Stelle, die durch den Wert des Programmzaehlers P der virtuellen CPU festgelegt ist.

Format (b) stellt vor dem Testlauf P auf den Wert a. Mit Format (c) wird gegenueber (b) noch ein temporaerer Unterbrechungspunkt b festgelegt, Format (d) legt zwei temporaere Unterbrechungspunkte b und c an, die nach Rueckkehr in den Debugger-Mode geloescht werden. Mit den Formaten (e) und (f) wird der Einsatzpunkt P aus der CPU uebernommen und ein bzw. zwei temporaere Haltepnkte b bzw. b und c angegeben.

Beim Testlauf wird jeder Durchlauf durch einen Passpunkt protokolliert und nach Durchlaufen eines Passpunktes mit Durchlaufzaehlerwert Eins oder Erreichen eines temporaeren Unterbrechungspunktes der Testlauf abgeschlossen. Abschliessend wird der Befehlszaehlerstand und der erreichte CPU-Zustand auf der Konsole ausgegeben.

Wird mit Praefix "-" gearbeitet (Format (g)), dann wird der Testlauf analog zu den Formaten (a) bis (f) durchgefuehrt. Unterdrueckt wird nur das Protokoll bei Durchgang durch die Passpunkte, die nicht zur Beendigung des Testlaufs fuehren, d.h. Durchlaufzaehler ungleich Eins.

Die gleichzeitige Belegung einer Adresse mit einem Passpunkt und einem temporaeren Unterbrechungspunkt sollte vermieden werden. Der Passpunkt hat bei der Abarbeitung die hoehere Paritaet.

#### 4.6.7.4. Schrittweise Abarbeitung mit Spur (T-Kommando)

Mit diesem Kommando wird das Testprogramm schrittweise abgearbeitet. Jeder Schritt wird protokolliert, indem die CPU-Zustaende auf der Konsole vor jedem Programmschritt ausgegeben werden. Die schrittweise Abarbeitung wird wie das G-Kommando behandelt, indem auf der Adresse nach dem anstehenden Befehl automatisch vom Debugger ein temporaerer Unterbrechungspunkt eingetragen wird und das G-Kommando (a) ausgefuehrt wird.
Bei Erreichen eines Passpunktes gilt hier die gleiche Vorrangregel wie beim G-Kommando.

Als Formate sind

(a) Tn
(b) T
(c) Tn,c
(d) -T....
(e) TW....
(f) -TW....

zugelassen.

Mit Format (a) werden n Schritte abgearbeitet, Format (b) entspricht Format (a) mit n=1.

Mit Format (c) wird vor jedem Eintritt bzw. Wiedereintritt in den Echtzeitbetrieb ein Unterprogramm abgearbeitet, das an der Adresse c beginnt. Parameter koennen an das Unterprogramm nicht uebergeben werden. Der Zustand der virtuellen CPU wird durch Abarbeitung dieses Unterprogrammes nicht veraendert.
Das Unterprogramm wird nach der Protokollierung des CPU-Zustandes und vor der zugehoerigen Protokollierung des anstehenden Befehls ausgefuehrt.
Die Erweiterung des Debuggers (Utilities) werden ueber diese Unterprogramme aktiviert.

Die Voranstellung des Praefix "-" vor die Formate (a) bis (c) schaltet die Substitution der protokollierten Operanden und Marken durch die symbolischen Referenzen ab (Formate (d)).

Die Formate (e) und (f) mit der Erweiterung "W" fuehren die Spurbildung so aus, dass Unterprogrammspruenge und ihre Ausfuehrung als ein abgearbeiteter Mikrobefehl gelten, d.h. alle Unterprogramme hoeherer Verschachtelung werden bei der Protokollierung ausgelassen. Die Voranstellung des Praefix "-" schaltet auch hier wieder die Abbildung der symbolischen Referenzen ab.

#### 4.6.7.5. Schrittweise Abarbeitung ohne Spur (U-Kommando)

Aehnlich dem Kommando T wird der Test schrittweise ausgefuehrt. Die Protokollierung wird nur vor Eintritt und nach Beendigung der Testschritte vorgenommen. Die Anzeige der Zwischenschritte wird unterdrueckt, ausser dem Durchlauf von Passpunkten.
Als Formate sind

(a) Un
(b) U
(c) Un,c
(d) -U....
(e) UW....
(f) -UW....

vorgesehen.

Die Bedeutung der Formate (a), (b), (c), (e) ist dem entsprechenden des T-Kommandos analog. Der Praefix "-" aber unterdrueckt nicht die Darstellung der symbolischen Referenzen, sondern unterdrueckt die Protokollierung der Passpunktdurchgaenge ausser dem Passpunktdurchgang mit Durchlaufzaehlerstand gleich Eins, der zur Beendigung des Testlaufes fuehrt.

## 4.7. Erweiterungen des Debuggers

Zur Steigerung der Leistungsfaehigkeit kann der Debugger durch Nachladen von Erweiterungsprogrammen aufgeruestet werden. Diese Programme sind vom Typ UTL und werden zur Erweiterung der Debuggerkommandos T und U herangezogen.

### 4.7.1. Arbeit mit den Erweiterungen

Der Lader im Debugger unterzieht Dateien vom Typ UTL einer Sonderbehandlung. Das Programm wird wie jedes andere Testprogramm in den TPA ab der Adresse 1ØØh eingeschrieben. Das Dienstprogramm laedt sich selbst an das obere Ende des TPA und besetzt den Bereich unmittelbar unterhalb des Debuggers. Gleichzeitig entsteht eine Symboltabelle, ueber die die Erweiterung bedient werden kann. Eine spaeter nachgeladene Symboldatei wird dann hier angekettet. Der TPA-Bereich wird entsprechend reduziert.

Ein Dienstprogramm hat 3 Eintrittspunkte zur Benutzung fuer die Initialisierung (INITIAL), fuer die Aufnahme von neuen Werten (COLLECT) und fuer die Anzeige der Statistik (DISPLAY).
Nach dem Laden eroeffnet der Debugger die Symboltabelle und meldet die Referenzen der Eintrittspukte fuer das Dienstprogramm

```
.INITIAL = IIII
.COLLECT = CCCC
.DISPLAY = DDDD
```

auf der Konsole.
Vor Benutzung der Erweiterung sind diese zu initialisieren. Waehrend des schrittweisen Testes von Programmen koennen dann Daten gesammelt werden, indem mit dem U- bzw. T-Kommando das Unterprogramm fuer Datensammlung

```
z.B. T7,.COLLECT
od.  U9,CCCC
```

angegeben wird. An jedem Haltepunkt werden dazu ueber das Unterprogramm Daten an das Dienstprogramm uebergeben, die dort erfasst und gesammelt werden. Dann kann die Statistik ueber die gesammelten Daten mit Aufruf eines Systemunterprogrammes auf der Adresse .DISPLAY bzw. DDDD in der Form

```
C.DISPLAY
```

bzw. CDDDD

ueber die Konsole ausgegeben werden. Mit dem Debuggerkommando

```
     C.INITIAL
bzw. CIIII
```

wird das Dienstprogramm wieder initialisiert.

### 4.7.2. Das Dienstprogramm HIST

Dieses Dienstprogramm erzeugt ein Histogramm in Balkendarstellung und ermittelt die relative Haeufigkeit des Ansprechens von Adressen im Test, gesammelt durch U- und T- Kommandos. Damit koennen die oft durchlaufenen Programmteile ermittelt werden. Nach Aufruf des Programmes wird automatisch das Initialisierungsprogramm ausgefuehrt. Es wird die Meldung

```
     "TYPE HISTOGRAM BOUNDS"
```

ausgegeben. Der Bediener muss zwei Parameter in der Form

```
     kkkk,gggg
```

mit der Bedeutung

```
     kkkk - untere Bereichsgrenze des Beobachtungsbereiches
     gggg - obere Bereichsgrenze des Beobachtungsbereiches
```

ueber die Konsole eingegeben. Anschliessend erscheinen die symbolischen Referenzen (siehe 4.7.1.) fuer die Eintrittspunkte auf der Konsole.

Dann koennen die Daten im Test gesammelt werden. Die Sammlung der Daten erfolgt ueber die Debuggerkommandos

```
     Tn,c   TWn,c   -Tn,c   -TWn,c
     Un,c   UWn,c   -Un,c   -UWn,c
```

jeweils vor dem Eintritt des Debuggers in den Testmodus.
Nach Abschluss der Sammelaktivitaeten kann das Histogramm durch den Unterprogrammaufruf

```
     C.DISPLAY
bzw. CDDDD
```

aufgerufen werden. Die Ausgabe erfolgt im Format

```
     HISTOGRAMM:
        ADDR     RELATIVE FREQUENCY, MAXIMUM VALUE = nnnn
        AAAA     *****
        BBBB     ***************
        ....
        ....
        ....
        YYYY     ***********************************
        ZZZZ     *************
```

Die Adressen AAAA bis ZZZZ stellen die Adressen innerhalb des definierten Beobachtungsbereiches dar. Der Wert nnnn ist das Maximum innerhalb der Tabelle. Die Saeulen innerhalb des Diagrammes werden durch die Anzahl der "*" dargestellt und sind durch den Wert nnnn normiert, die Skalierung wird ausserdem durch die Bildschirmbreite begrenzt.
Fuer eine Reinitialisierung muss das Initialisierungsprogramm angesprochen werden.

### 4.7.3. Das Dienstprogramm TRACE

Diese Erweiterung ist zur Protokollierung des Programmverlaufs unter Test vorgesehen. Dazu werden in einem zyklisch verwalteten Speicher die letzten durchlaufenen Programmpunkte gespeichert. Die Erweiterung muss vom Typ UTL sein. Beim Laden der Erweiterung wird der TPA zerstoert. Das Programm wird ab der 1ØØH geladen und aktiviert. Damit verschiebt sich die Erweiterung an das Ende des TPA, reduziert den TPA und fuehrt die eigene Initialisierung durch. Ausserdem wird ueberprueft, ob der Debuggerteil Asembler/Disassembler vorhanden ist, um die zuletzt abgearbeiteten Befehle auch in der mnemonischen Darstellung protokollieren zu koennen. Ist dieser Teil vorhanden, wird die Meldung:

```
READY FOR SYMBOLIC BACKTRACE
```

abgegeben. Damit werden dann auch alle anderen entsprechenden Debuggerkommandos durch die Disassemblierung unterstuetzt.
Anderenfalls wird die Meldung:

```
"-A" IN EFFECT, ADRESS BACKTRACE
```

abgegeben.

Die Initialisierung der Erweiterung wird mit der Anlage einer Symboltabelle fuer die TRACE-Erweiterung abgeschlossen. Diese Tabelle wird in der Form

```
INITIAL = IIII
COLLECT = CCCC
DISPLAY = DDDD
```

auf der Konsole angezeigt und stellt die zugelassenen Symbole und die entsprechenden realen zugehoerigen Speicheradressen dar. Diese stellen die Eintrittsadressen fuer das Initialisierungs-, Datensammel- und Anzeigeprogramm dar.

Nach Abschluss der Initialisierung ist der Spurspeicher geloescht und die Steuerung an den Debugger uebergeben. Die Anweisung fuer die Datensammlung (Erfassen der Spurpunkte) kann ueber die Anweisungen der U- bzw. T-Kommandos

```
Un,c   UWn,c   -Un,c   -UWn,c
Tn,c   TWn,c   -Tn,c   -TWn,c
```

aktiviert werden.
Fuer c muss hier die Unterprogrammadresse .COLLECT stehen. Dieses Programm wird immer vor Eintritt in den schrittweisen Testmode ausgefuehrt, d.h. diese Adressen werden in die Spurtabelle aufgenommen.

Es koennen auch Passpunkte waehrend der Datensammlung gesetzt werden. Erreicht der Durchlaufzaehler den Wert Null, fuehrt das zum Abbruch des Testmodus.

Mit dem Unterprogramm .DISPLAY wird das Spurprotokoll aus dem zyklischen Sammelspeicher auf der Konsole angezeigt.

Die Ausfuehrung des Unterprogrammes .INITIAL loescht den Datensammelspeicher und gibt die Adressen der Unterprogrammeintrittspunkte wieder auf der Konsole an. Danach kann eine weitere Datensammlung beginnen.